Pacotes RPM - A Ferramenta RPM no Gerenciamento de Sistemas GNU/Linux

Keynes Augusto de Deus

2006

Keynes Augusto de Deus

Pacotes RPM - A Ferramenta RPM no Gerenciamento de Sistemas GNU/Linux

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do curso de Pós-Graduação *Lato Sensu* em Administração de Redes Linux, para a obtenção do título de especialista em Administração de Redes Linux.

Orientador:
Prof. Joaquim Quinteiro Uchôa

LAVRAS
MINAS GERAIS - BRASIL
2006

Keynes Augusto de Deus

Pacotes RPM - A Ferramenta RPM no Gerenciamento de Sistemas GNU/Linux

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do curso de Pós-Graduação *Lato Sensu* em Administração de Redes Linux, para a obtenção do título de especialista em Administração de Redes Linux.

Aprovada em 28 de Setembro de 2006.

____
Prof. Heitor Augustus Xavier Costa

____
Prof. Simone Markenson Pech

____
Prof. Joaquim Quinteiro Uchôa
(Orientador)

LAVRAS
MINAS GERAIS - BRASIL

Resumo

O gerenciamento dos programas instalados em um sistema GNU/Linux é uma tarefa complexa e que exige muito do conhecimento dos administradores de rede. A utilização de um sistema de empacotamento de programas como RPM permite racionalizar os recursos empregados nessa tarefa. Ao mesmo tempo que trás benefícios para a segurança, integridade e a qualidade dos programas instalados desta forma. Entender o funcionamento deste mecanismo é importante para a formação dos profissionais em administração de redes Linux. Neste trabalho, será realizada a abordagem dos principais conceitos relacionados com a gestão e geração de pacotes de programas utilizando a ferramenta RPM. Tema discutido e exemplificado em seus variados aspectos através de uma abordagem voltada para a administração de redes de computadores baseadas em sistemas GNU/Linux.

Sumário

Lista de Quadros

1 - Introdução

Segundo (Guru Labs, 2005), a instalação de programas em sistemas do tipo Unix é tradicionalmente feita através da compilação de arquivos de código-fonte. O código-fonte é um arquivo salvo no formato de texto puro, cujo conteúdo são instruções que representam um programa. A compilação é um processo cujo objetivo é a geração de arquivos binários executáveis a partir da tradução do código-fonte para um formato executável.

Normalmente, as instruções de um programa são escritas na linguagem C ou C++. Essas instruções representam a codificação da lógica necessária para criar um programa de computador. O código-fonte, no estado em que se encontra, não é passível de execução por um computador, pois este não está preparado para interpretar diretamente as instruções do programa escritas nessa linguagem. Por isso, é necessário utilizar um compilador para gerar um programa executável em uma linguagem que a máquina interprete corretamente.

Um compilador é um conjunto de programas que realizam a transformação do código-fonte não executável em um código-objeto. Esse código-objeto é ligado às bibliotecas de funções de forma dinâmica ou estática, dando origem aos arquivos executáveis. As diversas bibliotecas formam uma coleção de funções pré-programadas e disponíveis para uso do programador na construção de programas.

Os arquivos de código-fonte são distribuídos geralmente no formato **tar.gz**. Esse formato de arquivo é simplesmente código-fonte empacotado por um utilitário como o **Tar**[1] e compactado no formato **Gzip**[2]. Após obter o código-fonte de um programa, é preciso descompactar e desempacotar o arquivo **tar.gz.** Somente após esses procedimentos é possível realizar a compilação. Na

1 Um utilitário para empacotamento de arquivos usado em sistemas GNU/Linux. Acrônimo de *Tape Archive.*

2 Um utilitário usado para compactar arquivos no sistema GNU/Linux.

compilação, são gerados arquivos executáveis que podem ser instalados no sistema e, enfim, serem executados pelo computador.

No GNU/Linux, esses processos envolvidos na compilação podem ser feitos manualmente através de comandos do operador no terminal e exigem amplo conhecimento do funcionamento do sistema.

Desta forma, para usuários menos experientes, esses processos são dificultadores na ampliação do uso de sistemas GNU/Linux. Obter os arquivos de código-fonte, escolher os locais de instalação, configurar e realizar a compilação são processos relativamente complexos.

Além disso, para realizar esse processo de compilação, o ambiente do sistema deve estar configurado com a presença de uma série de requisitos, tais como: arquivos de bibliotecas de desenvolvimento, compiladores, editores de ligação e outros programas envolvidos nos processos de compilação.

Deste modo, é exigido um esforço na administração do sistema para que seja possível trabalhar com arquivos de código-fonte. Além das questões relativas ao processo de compilação, sistemas que não utilizam uma forma de gerenciamento de pacotes são mais difíceis de serem atualizados e mantidos pelo administrador. A maioria dos usuários precisa usar o sistema GNU/Linux apenas para editar um texto, ler o correio eletrônico ou acessar a Internet. Esses usuários não devem se tornar especialistas em sistemas de computação apenas para conseguir utilizar seus aplicativos.

Mesmo administradores de sistemas podem perder muito tempo produtivo com esses detalhes. Seus empregadores podem não estar dispostos a pagar pelo tempo gasto para instalar e manter um grupo de programas manualmente. Essas características tornam a instalação, atualização e remoção de aplicativos em sistemas GNU/Linux um processo lento e bastante técnico.

Contudo, foram desenvolvidas ferramentas que possibilitam um melhor gerenciamento dos aplicativos instalados no sistema. Os aplicativos podem ser

distribuídos em um formato pronto para a instalação, o que elimina esses procedimentos manuais de instalação e a complexa operação de compilação. Esse formato de distribuição de aplicativos é chamado de empacotamento de aplicativos. Assim, um pacote é um arquivo cujo conteúdo é um aplicativo pronto para ser instalado em um sistema. Segundo (Nemeth, 2004), os formatos de pacotes mais bem sucedidos para o uso em sistemas GNU/Linux são: o RPM (*Red Hat Package Manager*) e o DEB (Debian *Package*).

Um dos objetivos deste trabalho é apresentar os conceitos sobre o formato de distribuição de aplicativos RPM e o uso da ferramenta de gerenciamento de pacotes rpm. Desta maneira, usa-se a grafia em maiúsculas para diferenciar o formato de pacote e minúsculas, para o aplicativo de gerenciamento. Assim, **rpm** se refere ao aplicativo de gerenciamento, enquanto **RPM** define um formato de distribuição de pacotes.

Outro objetivo deste trabalho é criar uma documentação que facilite o entendimento dos conceitos, da filosofia e dos processos envolvidos na gerência de um sistema baseado em pacotes RPM. Administradores de sistemas novatos gastam muito tempo na busca de conhecimento sobre o funcionamento do sistema de gerenciamento de pacotes. Reduzir o custo desta curva de aprendizagem é importante para a difusão da filosofia GNU/Linux e para a qualificação desses profissionais.

Este trabalho está estruturado do seguinte modo: no capítulo 2, são tratados os conceitos gerais e as definições da filosofia de gerenciamento de pacotes RPM; no capítulo 3, é abordado o tema da obtenção de pacotes RPM; o capítulo 4 enfoca tanto as pragas virtuais quanto o sistema de segurança utilizado por RPM para manter a integridade dos pacotes; no capítulo 5, é demonstrado como funciona a estrutura da base de dados de RPM; o capítulo 6 enfoca as operações com pacotes tanto na teoria como na prática; o capítulo 7 introduz a criação de pacotes RPM; o capítulo 8 trata do tema de ajuste do ambiente para a produção

de pacotes RPM; por fim, no capítulo 9, demonstra-se as opções de uso da ferramenta de construção de pacotes denominada `rpmbuild`. A leitura atenta desses capítulos fornece a base teórica e prática para a produção de pacotes RPM pelo leitor.

Por ser esse um tema muito extenso, fica inviável cobrir detalhes como: o uso de ferramentas gráficas usadas no gerenciamento de pacotes RPM; o funcionamento das APIs[3] fornecidas pela Red Hat e usadas nas linguagens Perl, C e Python. Também não serão abordados o uso avançado das ferramentas fornecidas pelo sistema RPM e aspectos ligados à programação de macros. Outros temas paralelos que necessitem de aprofundamento técnico e que por isso estão fora do escopo do tema principal poderão ser melhor compreendidos através das indicações de leitura e da bibliografia recomendada.

As explicações dos termos e dos conceitos sobre o sistema RPM serão feitas com ênfase no ponto de vista dos administradores de sistemas. Para tanto, será utilizado o ambiente baseado em caractere, apesar da existência de interfaces gráficas para os gerenciadores. Os exemplos práticos deste trabalho foram feitos em um sistema GNU/Linux Fedora Core 5 usando a versão 4.4.2 de RPM.

3 API – Segundo (Dígitro, 2006) é a sigla para *Application Program Interface,* um conjunto de rotinas, protocolos e ferramentas para a construção de aplicativos de software.

2 - Conceitos Gerais.

2.1 – Conceitos básicos sobre pacotes.

Na visão de (Foster-Johnson, 2005), um pacote é basicamente um arquivo compactado cujo conteúdo é um conjunto de arquivos, de diferentes formatos e tipos, e que juntos formam um aplicativo.

Apesar de possuir vários arquivos, um pacote é visto como uma unidade atômica; e, por isso, indivisível. Assim sendo, esse conjunto de arquivos deve necessariamente fazer parte de um mesmo aplicativo ou, no mínimo, estar ligado por similaridade de função. Em geral, os arquivos que formam um pacote são do tipo: binários executáveis, arquivos de configuração, *scripts* do *shell*, imagens, ícones, arquivos de som, arquivos de documentação e manuais. Porém, qualquer tipo de arquivo pode ser empacotado pelo sistema RPM.

Todavia, deve-se ressaltar que um pacote nem sempre é somente um aplicativo para ser instalado no sistema. Um pacote pode ser também uma coleção de arquivos necessários para a configuração do próprio ambiente do sistema. Assim, observa-se que a distribuição de arquivos no formato de pacote facilita também a configuração do próprio sistema GNU/Linux e não somente a distribuição de aplicativos ou programas utilitários.

Um exemplo deste fato é o pacote `rootfiles`[4] utilizado para configurar o sistema. Esse pacote não instala um aplicativo, mas alguns arquivos no diretório `/root`. Esses arquivos são similares aos arquivos do diretório `/etc/skel`[5], que são colocados nas pastas pessoais dos usuários comuns quando na criação desses usuários. No caso do pacote `rootfiles`, esses arquivos são especialmente ajustados para o uso do superusuário e instalados no momento da criação desta

4 Rootfiles é um pacote da distribuição FEDORA e pode ter um nome diferente para outras distribuições.

5 Basicamente, os arquivos são .bash_profile, .bash_logout e .bash_rc. Observe que são todos arquivos ocultos.

conta administrativa.

Diferentemente da instalação pelo processo manual, que exige a compilação do código-fonte, a instalação de um pacote binário dispensa esse processo. No pacote binário, os arquivos estão compilados e prontos para instalação. Os pacotes binários são totalmente ajustados para serem instalados na versão da distribuição GNU/Linux, para o qual foram criados. Isso torna um pacote construído para uma certa distribuição exclusivo da versão para o qual foi criado, apesar de haver exceções como os pacotes "sem arquitetura" ou mesmo criados para funcionar em múltiplas distribuições. Esta exclusividade pode ser vista como uma desvantagem no uso dos pacotes, pois na atual forma como são criados, a cada nova versão do sistema operacional é necessário reconstruir os pacotes novamente, adaptando-os à nova versão da distribuição.

Um dos principais motivos dessa incompatibilidade está relacionado com a questão da hierarquia dos diretórios. Apesar dos esforços de padronização promovidos por entidades como a LSB[6] através de documentos como o FHS[7], é comum que a hierarquia de diretórios seja levemente diferente entre as distribuições. O local de instalação de um programa na hierarquia do sistema de arquivos é uma das opções pré-definidas pelo empacotador do aplicativo. Como a hierarquia de diretórios de cada distribuição apresenta pequenas diferenças em relação às outras distribuições, as distribuições podem se tornar incompatíveis.

Esse fato gera incompatibilidades também no uso dos pacotes. Mesmo entre distribuições que utilizam um sistema de empacotamento de aplicativos idêntico, não há compatibilidade total. Além disso, a ferramenta de gerenciamento de pacotes pode ser implementada de maneira diferente entre as distribuições. Por essas razões, é recomendável, na maioria das vezes, utilizar pacotes criados especificamente para a versão da distribuição em uso.

---

6 LSB acrônimo para *Linux Standard Base*, entidade que define padrões abertos encontrada em www.linuxbase.org

7 FHS acrônimo para *Filesystem hierarchy Standard*, um protocolo para localização de arquivos e diretórios.

Outra opção previamente definida pelo empacotador são as permissões de segurança dos diretórios e arquivos a serem instalados. De acordo com (Silva, 2006), “as permissões de segurança são os atributos relativos à segurança do sistema de arquivos que impedem que usuários mal intencionados realizem operações indevidas com os arquivos”.

As operações mais comuns com arquivos são: ler, editar, gravar, apagar e copiar. Para realizar uma dessas operações, o usuário deve ser explicitamente autorizado. Desta maneira, as permissões de segurança definem o comportamento do sistema sempre que um usuário tenta acessar um arquivo para realizar uma dessas operações. São essas permissões que determinam se a operação é liberada ou bloqueada para o usuário que solicitou a operação. Para entender o completo funcionamento das permissões consulte (Silva, 2006).

Parte dos parâmetros pré-configurados no pacote podem ser personalizados pelo administrador no momento da instalação. Um exemplo é o diretório de instalação do aplicativo. Contudo, é recomendável não alterar essas pré-configurações e procurar usar a configuração padrão.

2.2 - O formato RPM de empacotamento.

Na avaliação de (Nemeth, 2004), entre os sistemas de empacotamento RPM (*Red Hat Package Manager*) e DEB (Debian), que são funcionalmente idênticos, "batalhas foram travadas para definir o melhor formato. Na prática, o formato RPM parece ter vencido a guerra".

Uma das razões dessa afirmação é, no ano de 2002, o formato RPM ter sido adotado pela *Linux Standards Base* como parte desse padrão. A versão 1.2 da especificação LSB define que RPM é o formato padrão para empacotamento de aplicativos em sistemas GNU/Linux compatíveis com esta especificação. Essa decisão torna RPM, anteriormente reconhecido como um padrão *de facto* para empacotamento de aplicativos GNU/Linux, também um padrão *de jure*. Essa padronização é importante para o desenvolvimento de sistemas livres, pois, sem essas regras, estes tendem para a incompatibilidade.

A empresa Red Hat distribuiu o sistema RPM com o licenciamento de código-aberto[8] (*open-source license*), motivo que facilitou a adoção do formato RPM como um padrão *de facto* por algumas distribuições. A adoção como um padrão *de jure* pela LSB revela o reconhecimento do bom desempenho do sistema. Segundo (LSB, 2002), "os aplicativos devem ser distribuídos no formato de empacotamento RPM conforme definido no apêndice de 1997 da edição do livro *Maximum* RPM".

A sigla RPM é formada a partir do nome, em inglês, do Gerenciador de Pacotes Red Hat (*Red Hat Package Manager*), como é chamado oficialmente.

8 A definição de Open Souce Iniciative (OSI) pode ser consultada na Internet em http://www.opensource.org/

2.3 - A nomenclatura de pacotes RPM.

Segundo (Foster-Johnson, 2005), o pacote RPM é um arquivo que, como qualquer arquivo, deve possuir um nome. Todavia, no caso dos arquivos de pacotes RPM, esses nomes não são aleatórios ou insignificantes. De acordo com (Guru Labs, 2005), a nomenclatura de um pacote RPM é padronizada e fornece uma série de informações sobre o aplicativo que está empacotado. A estrutura de nomes de arquivos de pacotes RPM possui o seguinte padrão:

```
NOME-VERSÃO-REVISÃO.ARQUITETURA.RPM
```

Deste modo, um nome de arquivo de um pacote RPM é formado por cinco partes básicas que são separadas umas das outras por traços ou pontos.

São constituintes do nome de arquivo de pacote RPM: o nome do aplicativo, a versão, a revisão, a arquitetura e a extensão de arquivo. Desta maneira, quando se avalia um nome de arquivo de um pacote RPM, é possível identificar detalhes desses componentes comuns na formação do nome. A partir dessas partículas que formam o nome pode-se extrair variadas informações sobre o pacote em questão. São exemplos válidos de nomes de pacotes:

- ✔ `kernel-smp-2.4.18-3.i586.rpm;`
- ✔ `ypbind-1.19-0.i386.rpm;`
- ✔ `yp-tools-2.9-0.i386.rpm;`
- ✔ `yum-2.6.0-1.noarch.rpm.`

O nome do pacote geralmente identifica a aplicação que está empacotada. No caso do pacote `ypbind-1.19-0.i386.rpm`, ele identifica uma conhecida aplicação denominada NIS (*Network Information Service*). O NIS era original-

mente chamado de páginas amarelas (*Yellow Pages*).

Contudo, essa era uma marca registrada da Sun Microsystems[9]; então, o nome foi alterado para NIS. Entretanto, o nome do pacote permaneceu com as iniciais `yp` por tradição. Neste caso, fica evidente que o nome do pacote pode ou não guardar uma similaridade com o nome do aplicativo que o mesmo implementa. Em geral, o nome do pacote é idêntico ao nome do aplicativo ou ao nome do serviço que o mesmo implementa, mas isso não é uma regra.

O nome do pacote costuma ser um parâmetro de pesquisa usado na realização de uma consulta à base de dados de pacotes RPM. Além do parâmetro "nome" é possível também incluir a versão e a revisão como critérios opcionais de consulta. Nesse caso, deve ser observada corretamente a grafia do nome do pacote, pois o mesmo pode ter um nome simples ou um nome composto.

Por exemplo, o pacote denominado `ypbind-1.19-0.i386` possui um nome simples, enquanto o pacote denominado `yp-tools-2.9-0.i386` possui um nome composto. Então, observa-se que, para separar nomes compostos, é usado o traço. Na formação de números de versão, é usado o ponto como separador. Deste modo, caso o nome do primeiro pacote seja usado como um parâmetro de consulta, escreve-se `ypbind`. Entretanto, para uma consulta tendo como parâmetro o nome do segundo pacote, este deve ser grafado como `yp-tools`, em função de ser um nome composto. Além disso, nomes de pacote são sensitivos quanto aos caracteres minúsculos e maiúsculos. A grafia desses nomes deve sempre diferenciar entre letras maiúsculas ou minúsculas.

Outra observação importante: nomes de arquivos de pacotes têm obrigatoriamente a extensão "*.rpm*". A extensão de arquivos está relacionada com os tipos MIME e com números mágicos que identificam esses arquivos no sistema.

Segundo (Tanenbaum, 2003), MIME é o acrônimo para *Multipurpose In-*

9 O endereço eletrônico da Sun Microsystems na Internet é www.sun.com.br/

ternet Mail Extensions, conceitos que são definidos nas RFC[10] 2045 a 2049. Basicamente, os tipos MIME definem categorias de arquivos distintas por suas extensões de arquivo. Extensões de arquivos são importantes, pois permitem associar arquivos aos programas utilitários com capacidade de manipulá-los. Deste modo, quando o sistema se depara com uma extensão registrada, ele pode disparar automaticamente um utilitário ou indicar ao usuário que abra o arquivo com um utilitário apropriado para manipular esse tipo de arquivo.

Segundo o manual do comando `file`, os números mágicos estão em um arquivo em `/usr/share/file/magic`. Esses são identificadores do conteúdo do arquivo, que podem ser consultados informando sobre o tipo do arquivo. Quando um utilitário tenta ler ou executar um arquivo associado, o número mágico é consultado para confirmar se a operação pode ser realizada.

Se nos nomes de arquivos é obrigatória a existência de uma extensão, em contrapartida, nomes de pacotes não têm extensão. Enquanto não instalados no sistema os pacotes devem ser vistos apenas como arquivos. Após sua instalação, a base de dados armazena informações apenas sobre pacotes e pacotes não possuem extensão. Deste modo, o arquivo `yp-tools-2.9-0.i386.rpm` é conceitualmente diferente do pacote implementado por ele, que, neste caso, é chamado `yp-tools-2.9-0.i386`.

Compreender a nomenclatura de pacotes e entender a diferenciação entre nomes de arquivos e nomes de pacotes é importante para entender o funcionamento das consultas à base de dados RPM. A seguir, apresenta-se um exemplo de consulta que utiliza o nome composto do pacote como parâmetro de pesquisa.

```
$ rpm -q yp-tools
yp-tools-2.9-0
```

10 Acrônimo para *Request for Comment.* São documentos que a descrevem segundo uma padronização técnica.

Essa consulta retorna como resposta o nome `(yp-tools)`, a versão `(2.9)` e a revisão `(0)` do pacote. É possível também consultar as informações sobre a arquitetura para a qual o pacote foi construído. Para isso, basta utilizar uma sintaxe mais complexa na consulta. Normalmente, essa informação é omitida, pois um pacote instalado somente pode pertencer a uma arquitetura compatível com a máquina consultada. No exemplo seguinte, a arquitetura consultada é `i386` denominada como Intel compatível 32-bit.

```
$ rpm -q yp-tools --qf "%{name}-%{version}-%{release}-%{arch}\n"
yp-tools-2.9-0-i386
```

Observando um nome de arquivo, tal como `kernel-smp-2.4.18-3.i586.rpm`, é possível extrair outros tipos de informações sobre o pacote além da versão, revisão ou arquitetura. Nesse exemplo, o nome `kernel-smp` indica que este é um pacote próprio para uso em computadores com suporte a SMP.[11]

Segundo (Tanenbaum, 2001), sistemas SMP são computadores que possuem mais de um processador, sejam estes reais (físicos) ou virtuais (lógicos), e, além disto, compartilham o mesmo barramento e a mesma área de memória RAM. Deste modo, baseado no nome do pacote, fica evidente que este arquivo deve ser instalado em um sistema com estas características. Este nome de arquivo indica que este *Kernel* possui características apropriadas aos sistemas SMP.

O segundo componente do nome de arquivo de um pacote é a versão. A versão é formada por uma série numérica que representa a evolução deste aplicativo ao longo de seu desenvolvimento. Um número de versão, normalmente, é incrementado quando um aplicativo recebe revisões importantes ou quando modificações tão significativas são feitas, que podem até torná-lo incompatível com suas versões anteriores.

11 *SMP - Symmetric MultiProcessing*, acrônimo para Multiprocessamento Simétrico.

Segundo (Guru Labs, 2005), o campo da versão deve conter somente números. A inclusão de letras não é aceita e irá causar erros nas ferramentas que manipulam estes nomes de pacotes. A indicação de número de versão é uma característica definida pelo desenvolvedor do aplicativo. Não existem regras padronizadas para quando um número de versão deve ser incrementado. Cada desenvolvedor define a numeração de versões de seu aplicativo do modo que lhe seja mais conveniente. Entretanto, este número de versão deve sempre ser crescente entre uma versão antiga e uma versão nova.

Por exemplo, o pacote `kernel-smp-2.4.18-3.i586.rpm` informa que a versão deste pacote é `2.4.18`. Deste modo, se identifica um *kernel* da série `2.4`. Segundo (Ribeiro, 2004), "os números de série de pacotes do *kernel* terminados com números pares indicam um *kernel* estável. Os números de série terminados com ímpares indicam um *kernel* em desenvolvimento". Uma utilidade para esses números de série é caracterizar funcionalidades presentes no aplicativo. Desta maneira, um pacote da série `2.4` pode não ter características funcionais presentes na série `2.6`. Assim, o nome da versão é importante também para que o administrador possa definir o uso dos pacotes em função das novas tecnologias contidas neles.

O terceiro componente do nome de arquivo de um pacote é a revisão ou *release.* O termo inglês *release* significa liberação. Basicamente, esse termo caracteriza a liberação de uma revisão. Desta maneira, se considera a liberação de um pacote como uma revisão. Afirma (Guru Labs, 2005) que "a revisão é o indicativo de que mesmo após os testes realizados com uma versão ou mesmo após o empacotamento de uma versão, foram necessários pequenos ajustes ou correções no aplicativo ou mesmo no pacote". Assim, a versão do aplicativo permanece a mesma, mas uma revisão foi feita para corrigir problemas com o aplicativo ou com o pacote. As revisões também podem indicar que o desenvolvedor

aplicou uma nova atualização[12] (*patch*) desenvolvida para aquela versão.

O quarto componente do nome de arquivo de um pacote é a arquitetura. A arquitetura identifica o tipo de processador para o qual o pacote foi construído. Pacotes construídos para um determinado tipo de CPU[13] podem não funcionar plenamente em outro tipo de CPU. Isso ocorre porque, durante o processo de construção do pacote, características específicas existentes em uma CPU são ativadas no processo de compilação do aplicativo. Deste modo, um pacote construído para uma arquitetura específica pode utilizar recursos ou funções próprias dessa arquitetura com maior desempenho.

Outro ponto importante é diferenciar os conceitos de **plataforma de computadores** de **arquitetura de processadores**. Segundo (Intel, 2006), "arquitetura é o projeto básico de um microprocessador. Pode englobar a tecnologia de processos e/ou outros aperfeiçoamentos da arquitetura".

Por exemplo, os processadores Pentium 3 e Pentium 4, ambos produzidos pela Intel, possuem diferentes **arquiteturas**, ou seja, seus projetos básicos são diferentes.

De acordo com as definições de (Foster-Johnson, 2005), plataforma define uma família de processadores que apresentam arquiteturas diferentes entre si, mas que, entretanto, mantêm uma similaridade mínima para formarem uma plataforma. Por exemplo, a plataforma denominada **Intel compatível - 32 bits**, apresenta uma série de famílias de processadores como `i386, i486, i586 e i686`. Estes, por sua vez, podem ou não ser do mesmo fabricante, mas certamente são da mesma **plataforma,** apesar das **arquiteturas** diversas. Outros exemplos: a família de processadores Duron produzidos pela AMD[14] e a família

12 Segundo (Dígitro,2006), em computação, um *patch* é uma atualização de software para correção de problemas (*bugs*) de um programa ou para melhorar a performance ou usabilidade deste.

13 Segundo (Dígitro, 2006), CPU é o acrônimo para *Central Processing Unit*, é o cérebro do computador, também referido como processador ou processador central.

14 AMD é um concorrente da Intel na produção de processadores. Pode ser localizado em www.amd.com.br.

de processadores Celeron produzidos pela Intel são ambos da mesma plataforma. Todavia, eles são de fabricantes distintos e têm arquiteturas também distintas. Mesmo assim, pertencem à mesma plataforma denominada Intel compatível - 32 bits, pois possuem um conjunto mínimo de instruções que os tornam compatíveis. Ambos podem ser utilizados para executar um sistema operacional como o GNU/Linux compilado para a plataforma Intel compatível - 32 bits, mas não podem executar o GNU/Linux compilado para outra plataforma, por exemplo, Power PC.

Desta forma, a escolha de pacotes deve sempre ser compatível com a arquitetura do processador onde serão executados. O desempenho dos aplicativos desses pacote será maior em função de terem sido construídos objetivando o uso de características especiais de um determinado processador. Processadores da arquitetura `i686`, por exemplo, têm capacidade para executar programas compilados para as arquiteturas `i386, i486, i586 e i686`. A cada nova arquitetura lançada por um fabricante, as características das arquiteturas anteriores são englobadas e expandidas.

No caso da instalação de um pacote construído para a arquitetura `i386` em um computador cujo processador seja da série `i686`, diversos recursos avançados presentes nesse tipo de CPU e que poderiam ser utilizados podem ter sido desativados na compilação do aplicativo. O impacto no desempenho será sentido durante a execução do aplicativo ou mesmo em características funcionais do aplicativo que podem ser inclusive completamente desativadas.

Determinados aplicativos podem exigir uma certa arquitetura para serem totalmente funcionais; isso ocorre caso façam uso de instruções exclusivas dessa arquitetura. É comum que algumas distribuições somente disponibilizem pacotes para a arquitetura `i386`. Esse tipo de pacote da arquitetura `i386` pode ter um menor desempenho em sistemas com processadores mais modernos ou não utilizar novos recursos dessas CPUs, como afirmado anteriormente. Este tipo de pacote

é considerado genérico e são feitos para serem executados nas arquiteturas da plataforma Intel compatível - 32 bits, o que representa uma sub-utilização dos recursos dos processadores mais modernos.

O Quadro 1 relaciona os tipos de plataforma e de arquitetura indicados por (Foster-Johnsson, 2005), para as quais é possível criar pacotes RPM[15]. Neste Quadro, destaca-se a arquitetura denominada `noarch`. Pacotes, cuja arquitetura seja nomeada como `noarch`, indicam que se trata de um pacote construído de modo que, teoricamente, seja independente da arquitetura.

Afirma-se teoricamente, uma vez que, dependendo do que este pacote implementa, a transição entre plataformas pode exigir ajustes. Normalmente, pacotes denominados "sem arquitetura" implementam documentação, além de, algumas vezes, implementar também *scripts* do *shell* ou programas escritos em uma linguagem interpretada como Perl. Neste tipo de pacotes, no campo referente a arquitetura lê-se `noarch`. Por exemplo, `yum-2.6.0-1.noarch.rpm` é um nome de um pacote que é independente da arquitetura do processador para ser instalado. O aplicativo Yum é escrito na linguagem interpretada Python. Esse pacote será instalado em uma arquitetura que tenha o sistema Python instalado.

O quinto elemento componente do nome de arquivo de um pacote é a extensão. Deste modo, arquivos de pacote RPM possuem um nome terminado com a extensão de arquivo "*.rpm*". Arquivos de pacote possuem extensão, nomes de pacotes não a possuem. Deste modo, `xsane-gimp-0.99-2.2.i386.rpm` é o arquivo RPM do pacote `xsane-gimp-0.99-2.2.i386`.

---

15 Algumas arquiteturas podem não estar citadas nesse quadro devido aos constantes lançamentos de processadores.

Quadro 1 - Plataformas e arquiteturas suportadas por RPM.

Plataforma	Arquiteturas
Intel compatível 32-bit	i386, i486, i586, i686, athlon
Intel compatível 64-bit	ia64
HPAlpha	alpha, alphaev5, alphaev56, alphapca56, alphaev6, alphaev67
Sparc/Ultra Sparc (Sun)	sparc, sparcv9, sparc64
ARM	armv3l, armv4b, armv4l
MIPS	mips, mipsel
Power PC	ppc, ppciseries, ppcpseries, ppc64
Motorola 68000 series	m68k, m68kmint
IBM RS6000	rs6000
IBM S/390	i370, s390x, s390
Independente.	**noarch**

2.4 – As seções de um pacote RPM.

Segundo (Foster-Johnson, 2005), "um arquivo do tipo pacote RPM é composto internamente por quatro seções." Sendo nesta ordem: área de identificação (*lead*), assinatura (*signature*), cabeçalho (*header*) e conteúdo (*payload*).

A área de identificação é construída de acordo com a versão de RPM usada para construir o pacote. Pode ser chamada também de *lead* , termo inglês cujo sentido é precedência. A assinatura aparece após o *lead* e tem como função armazenar uma assinatura digital, gerada a partir de criptografia de chave pública ou a partir de um sumário de mensagens como MD5[16], cuja função é garantir a integridade do pacote e de seus arquivos.

O cabeçalho se constitui de blocos de dados que funcionam como uma etiqueta para fornecer informações sobre o pacote. São encontradas nessa seção informações sobre, por exemplo, número de versão, descrição do pacote, e tipo de licenciamento.

O conteúdo (*payload*) é a seção que armazena os arquivos a serem instalados pelo pacote. Essa seção é responsável pela "carga útil" de um pacote, pois os arquivos contidos nessa seção são instalados quando um pacote é instalado. Arquivos nessa seção são compactados no formato gzip. Uma vez descompactados, esses arquivos estão no formato de arquivo compatível com `cpio`, podendo ser extraídos com o utilitário `rpm2cpio`.

16 Segundo (Tanenbaum, 2003), MD5 é o acrônimo para *Message Digest 5*, ou sumário de mensagens versão 5. É um tipo de checagem para validar arquivos e garantir a integridade de seu conteúdo.

3 – Pacotes RPM e SRPM: Detalhes e formas de uso

3.1 – Os tipos de pacotes RPM e SRPM.

Segundo (Foster-Johnson, 2005), são dois os tipos de pacotes RPM que podem ser usados por um administrador: os pacotes binários conhecidos por RPM e os pacotes fonte chamados de SRPM. Cada um destes possui funções distintas e também são diferentes quanto ao modo de trabalhar com eles.

Segundo (Guru Labs, 2005), os pacotes binários são aqueles que normalmente são usados para instalar um aplicativo diretamente no sistema. Segundo (Foster-Johnson, 2005), esses pacotes são uma aplicação completa ou bibliotecas de função compiladas para uma arquitetura particular. Esses pacotes binários são aplicações que podem, ou não, depender da instalação de outros pacotes para serem executados.

Esses arquivos que fornecem funções para outros programas são chamados de dependências e são distribuídos na forma de pacotes binários. Normalmente, uma dependência é uma biblioteca, mas não obrigatoriamente, podendo ser um aplicativo qualquer, necessário para o funcionamento de outro programa. Deste modo, pode-se observar que pacotes binários possuem programas que são previamente compilados e estão prontos para serem instalados em uma arquitetura específica. Em contrapartida, pacotes fonte contêm a aplicação no formato de arquivo de código-fonte.

De acordo com (Guru Labs, 2005), o sistema RPM pode trabalhar com arquivos de pacotes fonte denominados SRPMs – *Source RPMs*, usando-os para construir um pacote binário para uma arquitetura específica.

Assim, como citado anteriormente, as distribuições GNU/Linux são formadas por pacotes genéricos para a arquitetura `i386`. Desta maneira, não aproveitam as funcionalidades dos processadores de última geração e não utilizam

muitos dos recursos desses processadores. Surge, neste caso, uma necessidade de gerar pacotes para uma arquitetura alvo específica, como `i686` ou `Pentium 4`, que fornecem opções de maior desempenho para os programas. Primordialmente, esta é a função dos pacotes SRPMs. Segundo (Guru Labs, 2005), um pacote SRPM é formado tipicamente por:

- O código-fonte original de uma aplicação (*Pristine Source*);
- As atualizações (*patches*) para este aplicativo;
- Arquivos auxiliares, como os *scripts* do tipo *System V Init* e outros para o uso de sistemas, como **Cron** ou **PAM**[17];
- Um arquivo do tipo *.spec*, formado por macros RPM e comandos do *shell* necessários para criar o pacote binário a partir do conjunto de arquivos de código-fonte original, *patches* e *scripts* de suporte.

Por convenção, os pacotes fonte SRPM possuem uma nomenclatura de arquivo idêntica a dos pacotes RPM binários. A única diferença é que ao nome do arquivo é acrescida a partícula "*.src*" antes da extensão de arquivo "*.rpm*". Assim, um nome de pacote fonte como `bash-3.1-6.2.src.rpm` é um nome de arquivo corretamente formado e válido para um pacote fonte.

Nem sempre existirá um pacote fonte do tipo SRPM correspondente a um pacote RPM binário. Uma das razões para isso é nem todo aplicativo empacotado no formato RPM ser um aplicativo de código livre (*open source*). Nestes casos, apenas os pacotes no formato binário RPM podem ser encontrados. O administrador não poderá fazer modificações no código-fonte dos programas desse pacote, pois este é um aplicativo de código fechado.

17 Cron é um utilitário para o agendamento de tarefas - PAM - *Pluggable Authentication Modules for Linux*

3.2 – Obtenção de pacotes RPM e SRPM.

Um sistema operacional como o GNU/Linux está em constante evolução. Novas versões de programas surgem quase que diariamente, seja para consertar falhas de segurança encontradas no código de algum programa ou para aprimorar a funcionalidade de uma aplicação.

Por esta razão, mais cedo ou mais tarde um sistema GNU/Linux sofrerá mudanças e será preciso fazer alterações nos programas instalados. Essas alterações são basicamente a instalação, a atualização ou a remoção de algum pacote do sistema. Deste modo, todo administrador deve obter os pacotes atualizados para realizar a manutenção de seu sistema.

A fonte primária de pacotes é formada pelos discos de instalação. Estes podem ser um conjunto de CDs-ROM ou um DVD[18], mas estes pacotes ficam rapidamente desatualizados. Assim, logo após a instalação de um sistema GNU/Linux, através dos discos de instalação, é preciso obter as atualizações de segurança e de aprimoramento.

Atualmente, os principais sistemas GNU/Linux são dotados de um mecanismo de atualização. Segundo (Morimoto, 2005), os mais conhecidos programas usados para atualização de sistemas são: o **apt-get** (*Advanced Packaging Tool*), usado pelos sistemas Debian e seus derivados; o **urpmi** usado pelo Mandriva; o **yum** (*Yellowdog Updater Modified*) usado pelo Fedora; e o **emerge** usado pelo Gentoo. O apt-get portado para sistemas RPM, sendo usado em distribuições como Conectiva. A partir da disponibilidade de um sistema de atualização, esta passa a ser a fonte secundária para obtenção de pacotes, seja para instalação ou atualização.

Atualmente, o sistema de atualização usado pelo Fedora Core é o Yum[19]

---

18 CD-ROM - Disco compacto somente de leitura. DVD - *Digital Versatile Disc.*

19 A página oficial do sistema yum fica em http://linux.duke.edu/yum.

que é fortemente acoplado ao RPM para encontrar, baixar e atualizar ou instalar um pacote. O sistema Yum pesquisa e resolve as dependências de um pacote antes de sua instalação ou atualização. Para isso, o sistema utiliza o conceito de repositórios de pacotes.

Repositório de pacotes é uma coleção de pacotes dividida em diversas categorias, normalmente disponíveis na Internet. O repositório é responsável por armazenar os pacotes base, que originalmente formam uma distribuição, e os pacotes referentes às atualizações do sistema. Repositórios também podem conter pacotes extras ou não-oficiais, mantidos por desenvolvedores particulares.

Em resumo, o repositório é o local mais indicado para procurar por um pacote para sua distribuição. Os repositórios fornecidos pelo distribuidor são chamados repositórios oficiais. Os repositórios-espelho chamados também de *mirrors*[20] são cópias idênticas dos repositórios oficiais. A função dos *mirrors* é distribuir geograficamente a carga no uso de repositórios. Isso é alcançado replicando o conteúdo dos repositórios oficias em vários outros.

Deste modo, o tráfego dos repositórios oficiais não sobrecarrega a Internet nem o repositório oficial. Repositórios são recursos escassos; seu uso deve ser o mais racional possível. É preciso conscientizar os usuários quanto a essas características dos repositórios para evitar desperdício de largura de banda com o tráfego exagerado de pacotes pela Internet. É possível criar repositórios particulares para uso de uma rede local. Outra forma é manter *caches* de pacotes; desta forma, os pacotes baixados anteriormente são reaproveitados.

No Fedora Core, o repositório responsável pelas atualizações é denominado **updates**. Os arquivos desse repositório são atualizações de programas distribuídos na relação de pacotes chamados "oficiais". Entretanto, nem todos os pacotes mais populares são suportados pelo distribuidor como pacotes oficiais.

Desta maneira, outro repositório muito popular é o repositório **extras**.

20 O termo inglês *mirrors* significa espelhos.

Nele, são armazenados pacotes "não-oficiais", que são distribuídos e mantidos por membros da comunidade de desenvolvedores.

Outra opção de localização de pacotes são os desenvolvedores que disponibilizam seus próprios repositórios particulares. Estes, por sua vez, podem estar fora do formato usado pelo programa de atualização. Neste caso, ainda é possível baixar e instalar esses pacotes diretamente utilizando o próprio sistema RPM como um cliente FTP ou HTTP.[21]

Assim, apesar da existência de aplicativos para manipular a atualização de pacotes, o sistema RPM também pode ser usado para essa finalidade. Entretanto, o uso de RPM como um sistema de atualização requer mais conhecimento por parte do administrador, pois rpm não é capaz de resolver as dependências e não atualizará (*update*) o sistema em um único comando como o Yum ou o apt-get são capazes de fazer.

Caso um administrador esteja procurando por um pacote que não encontrou nos repositórios oficiais, extras ou de terceiros, que sejam de seu conhecimento prévio, é possível utilizar as ferramentas de busca da Internet, como o Google[22] ou recorrer aos *sites* especializados em fornecer pacotes.

De acordo com (Uchôa; Simeone; *et al.,* 2003), existem vários *sites* especializados na divulgação e distribuição de pacotes. Merecem menção o Souce-Forge, Freshmeat, e o RPMFind.[23] Este último é um sistema de buscas específico e muito eficiente para encontrar pacotes RPM, podendo filtrar pacotes de acordo com vários critérios, como distribuição, arquitetura e nome do pacote.

21 FTP e HTTP são protocolos usados para transferência de arquivos ou páginas da Web.

22 Google é uma ferramenta de busca da Internet disponível em www.google.com.br

23 Encontrados em: www.souceforge.net, www.freshmeat.net e www.rpmfind.net

3.3 – RPM como um cliente FTP e HTTP.

O sistema RPM pode realizar funções como um cliente FTP ou HTTP para obter pacotes. As operações de consulta de informações do cabeçalho, instalação ou atualização podem ser feitas em pacotes armazenados em locais externos como um servidor FTP ou HTTP disponível na Internet. As transferências usando o protocolo FTP são feitas de modo passivo (*FTP PASV*), e, nos casos em que não são indicados um usuário e uma senha, é usado, por padrão, acesso anônimo. A diferença na sintaxe no uso de RPM como uma ferramenta FTP é, no lugar de indicar uma rota até o pacote, indicar um URL[24]. A sintaxe para utilizar URLs como caminho até o pacote é a seguinte:

```
ftp://USER:PASSWORD@HOST:PORT/caminho/para/o/pacote.rpm
```

A maioria dos repositórios de pacotes aceita acesso anônimo. Deste modo, as opções de segurança como usuário e senha são opcionais. Nos casos em que essas informações são obrigatórias e a senha seja omitida, um *prompt* será exibido para que a senha seja digitada em tempo de execução.

O hospedeiro dos pacotes pode ser indicado através de seu nome de domínio totalmente qualificado (FQDN[25]), por exemplo, `ftp.fedora.com`, sendo que esse endereço deve ser resolvido por um servidor de nomes de domínio (DNS[26]). Também é possível indicar diretamente um endereço IPv4 (*Internet Protocol versão 4*) na forma decimal, por exemplo `200.245.64.1`.

Normalmente, RPM tenta acesso ao serviço utilizando a porta TCP padrão. No caso de servidores FTP, `21` é a porta padrão. Os servidores HTTP utili-

24 URL (*Uniform Resource Locator*) é um ponteiro que indica uma informação específica disponível na Internet. As URL são comumente conhecidas como endereços eletrônicos.

25 FQDN – Acrônimo para *Fully Qualified Domain Name* ou Nome de Domínio Totalmente Qualificado.

26 DNS – Acrônimo para *Domain Name System* ou Sistema de Nomes de domínios.

zam a porta 80 como padrão para seu funcionamento. Caso o serviço seja oferecido em uma porta que não seja padrão, é necessário especificar também qual é esse número de porta. O caminho até o pacote pode incluir uma série de diretórios e sub-diretórios, além do nome do próprio pacote. Essas rotas e nomes devem ser corretamente grafados, levando-se em consideração maiúsculas e minúsculas, bem como incluir a extensão do arquivo desejado.

Segundo (Ewing; *et al*, 2002), caso seja usado um servidor *proxy*[27] para FTP ou HTTP, as informações sobre o endereço e o número da porta do servidor *proxy* podem ser especificadas do seguinte modo:

--ftpproxy *HOST*

O host definido em *HOST* será usado como um servidor PROXY para as transferências, o que permite usar FTP através de um FIREWALL que usa um sistema de PROXY. Essa opção pode ser configuradas através da macro **%_ftpproxy**.

--ftpport *PORT*

O número da porta TCP definida na opção *PORT* é usada caso um servidor PROXY esteja sendo usado. Esta opção pode ser definida na macro **%_ftpport**.

--httpproxy *HOST*

O host definido em *HOST* será usado como um servidor PROXY para as transferências usando o protocolo HTTP. Essa opção pode ser definida na macro **%_httpproxy**.

--httpport *PORT*

O número da porta TCP que será usada para a conexão é definido em *PORT*. **Essa opção pode ser especificada na macro %_httpport**.

27 O serviço de *proxy* ou encaminhamento é fornecido por sistemas como o Squid ou mesmo o próprio Apache.

4 – Mecanismos de segurança de RPM.

4.1 – As pragas virtuais e a segurança dos pacotes RPM.

Apesar dos sistemas GNU/Linux serem reconhecidos pelo ótimo nível de segurança que oferecem e pelo baixo número de pragas virtuais que atacam esses sistemas, é sabido que nenhum sistema é imune a falhas. Normalmente, as falhas de programação são as mais exploradas por pragas virtuais. São variados os tipos de ataques existentes. Os únicos pontos que esses ataques têm em comum são: visam cometer uma fraude eletrônica ou burlam a segurança do sistema para prover acesso ao sistema para que invasores possam realizar fraudes.

Segundo (Tanenbaum, 2001), uma categoria de ataques aos quais os sistemas GNU/Linux são bastante vulneráveis são os chamados “ataques de dentro do sistema”. Dentre os vários tipos de ataques desta categoria, destaca-se o ataque denominado *Trojan Horse* ou cavalo de Tróia. O sugestivo nome dado a esse tipo de ataque descreve o seu modo de operação que pode ser resumido da seguinte maneira: Juntamente com um programa aparentemente inocente, como um jogo interessante, está embutido um código malicioso que recebe o nome genérico de cavalo de Tróia. Ao ser executado, esse código gera um comportamento inesperado dentro do sistema, podendo agir de variados modos.

As pragas virtuais atuam roubando informações, adulterando partes do sistema, apagando dados, abrindo brechas na segurança ou danificando totalmente o sistema forçando a sua reinstalação. As pragas virtuais são hoje uma realidade no cotidiano da administração de sistemas de computadores e redes. Os administradores devem estar sempre alerta para evitar e identificar esses ataques quando acontecem, fazendo da segurança um item primordial.

De acordo com (Uchôa, 2003), código malicioso é um programa criado com finalidades mal intencionadas, como facilitar uma invasão ou furtar dados.

São exemplos de categorias de programas maliciosos os vírus de computador, os programas cavalo de Tróia (*Trojan Horse*) e os programas vermes (*Worms*). Como afirma (Uchôa, 2003), a maior parte dos programas maliciosos existentes são híbridos dessas três categorias de pragas. Desta maneira, eles podem explorar as melhores características de cada um desses tipos de praga virtual. Em conseqüência, essas pragas virtuais (vírus) seguem se propagando através de redes como a Internet, aproveitando-se das falhas em sistemas para se instalar (vermes) ou ficam ocultas dentro dos sistemas (cavalos de Tróia) para cometer ilícitos eletrônicos.

Deste modo, invasores de sistemas computacionais, inicialmente, podem distribuir seu código malicioso incluído em pacotes de programas. Mais tarde, ao serem executados, esses programas iniciam ou facilitam a execução de um ataque. Isso é possível, pois o código malicioso pode criar verdadeiros "buracos" na segurança do sistema expondo falhas que levam a um ataque remoto.

Programas distribuídos no formato de pacotes, como os RPM, são excelentes locais para esconder essas pragas virtuais. Como os sistemas GNU/Linux são baseados, em sua grande maioria, em programas de código aberto, é possível introduzir mudanças no código-fonte, compilar e distribuir pacotes adulterados.

Pensando exatamente na segurança dos sistemas em relação à possibilidade de injeção de código malicioso dentro de pacotes, foram criados mecanismos para garantir a integridade dos pacotes e dos arquivos distribuídos através de pacotes RPM. Essa segurança é totalmente baseada na tecnologia da criptografia de chave pública e nos algoritmos de sumários de mensagens.

Segundo (Tanenbaum, 2003), o sistema de criptografia baseado em chave pública proposto por Diffie e Hellman em 1976, era uma idéia radicalmente nova para a época. Pois, as chaves de criptografia e descriptografia eram diferentes e uma não era derivada da outra. Além disso, o algoritmo utilizado para codificar as informações também é público. Esse sistema exige que cada usuário seja

detentor de um par de chaves: uma chave pública e outra privada. A chave pública é distribuída para qualquer interessado em estabelecer uma comunicação segura com o dono da chave, sendo esta usada para gerar uma mensagem para o seu dono. A chave privada, de conhecimento apenas de seu dono, é usada para descriptografar as mensagens codificadas com sua chave pública.

Desta maneira, é possível estabelecer um sistema de assinaturas baseado em criptografia de chave pública. "O padrão *de facto* para os algoritmos de chave pública é o RSA (Rivest, Shamir, Adleman) que permanece resistindo às tentativas de rompimento por mais de vinte e cinco anos e é considerado um algoritmo muito forte", como afirma (Tanenbaum, 2003). Uma desvantagem do RSA é o tamanho mínimo da chave para garantir um nível desejável de segurança ser de 1024 bits; isso o torna lento para a codificação de mensagens grandes.

Deste modo, como demonstra (Tanenbaum, 2003), o sistema de criptografia de chave pública garante o sigilo e a integridade dos dados. Todavia, o nível de consumo computacional para usar RSA ou outro algoritmo de criptografia é muito grande tornando seu uso restrito a algumas operações críticas.

Em função disso, surgiram os sumários de mensagem[28], por exemplo, o MD5 (*Message Digest 5*) e o SHA1 (*Secure Hash Algorithm 1*). Segundo (Tanenbaum, 2003), os sumários de mensagem não se preocupam com a questão do sigilo das informações, entretanto são extremamente eficientes em relação à integridade dos dados. Sumários de mensagens podem ser vistos como somatórios de checagem (*checksum*). Um algoritmo matemático é aplicado utilizando cada um dos *bits* que formam a mensagem. Assim, a partir deste conteúdo, é gerada uma assinatura digital única chamada *hash*[29]. A alteração em apenas um único *bit* da mensagem gera um *hash* bem distinto do original.

Outra vantagem dos sumários de mensagem é o fato do *hash* gerado não

28 Os sumários de mensagem são genericamente denominados MD acrônimo para *Message Digest*.

29 É sabido que um *hash* pode se repetir, mas a probabilidade disso acontecer é de 2^{128} sendo considerada nula.

guardar qualquer referência em relação à mensagem que foi introduzida para ser assinada. Em função disso, não é possível deduzir uma mensagem à partir do *hash* criptográfico gerado pelo sumário de mensagem. Ou seja, o *hash* é unidirecional e independente, podendo ser transmitido separadamente da mensagem.

Deste modo, o receptor possui a mensagem e a assinatura *hash* gerada à partir dela; basta então confrontar um com o outro novamente para conferir a integridade do conteúdo. Caso haja divergência, mesmo que de um único *bit*, o *checksum* irá acusar a alteração gerando um *hash* diferente do original.

Deste modo, usando menos recursos computacionais que os processos criptográficos, é possível garantir a integridade de arquivos utilizando sumários de mensagens. O sistema RPM utiliza os dois métodos: assinaturas GPG de chaves públicas para autenticar pacotes e sumários de mensagens MD5 ou SHA1, para arquivos. Ambas informações são armazenadas no pacote e, após instalado, essas informações também são mantidas na base de dados para futuras consultas e verificações de integridade e autenticidade dos arquivos no sistema de arquivos.

4.2 – Importação de uma chave pública.

O sistema GNU/Linux que faz uso de pacotes RPM, na maioria das vezes, instala as chaves públicas do distribuidor. No Fedora Core, estas chaves estão armazenadas no diretório `/etc/pki/rpm-gpg`. Entretanto, rpm não faz a importação dessas chaves automaticamente. Assim, força uma importação manual das chaves públicas para que se possa trabalhar com os repositórios oficiais.

Arquivos de chaves públicas são gerados por programas específicos para lidar com criptografia, tais como o **PGP** (*Pretty Good Privacy*) ou **GnuPG**.[30] Quando exportada, uma chave é vista como um bloco de texto puro inserido em uma "armadura" de caracteres do tipo `ASCII`.

Por exemplo, o arquivo da chave pública do repositório **`extras`** do Fedora Core é instalado no diretório `/etc/pki/rpm-gpg/`, cujo nome é `RPM-GPG-KEY-fedora-extras` e têm como conteúdo uma chave pública. Este arquivo pode ser usado para validar pacotes obtidos, seja via FTP ou HTTP, deste repositório "não-oficial".

Entretanto, para que rpm possa validar os pacotes obtidos através de repositórios de terceiros ou "não-oficiais", é preciso importar a chave pública disponibilizada pelo empacotador. Em sistemas Fedora Core 5, importar a chave é equivalente a instalar um pacote chamado **gpg-pubkey-x**, onde x é o identificador da chave. Após o processo de importação, uma chave pode ser consultada ou excluída usando os mesmos recursos disponíveis para manipular pacotes.

Na maioria das vezes, a chave pode ser obtida diretamente do *site* do repositório, nos discos de instalação ou através de pacotes RPM criados com a finalidade de distribuir arquivos de chaves públicas.

30 Informações sobre GPG podem ser obtidas em http://www.gnupg.org/documentation. PGP usa o algoritmo IDEA (*International Data Encryption Algorithm*) que é patenteado. Por esta razão, não é considerado totalmente software livre. Deste modo, GPG é mais comum de ser encontrado nas distribuições GNU/Linux, já que utiliza o OpenPGP, que somente utiliza algoritmos não patenteados ou licenciados sob GPL.

Esta última opção, por exemplo, está disponível para o repositório independente Livna[31]. Este é um repositório "não-oficial" para o Fedora Core que mantém uma série de pacotes bastante populares; entre eles estão os *drivers* proprietários da Nvidia e da ATI, usados para dar suporte 3D às placas de vídeo desses fabricantes. Esses são arquivos que não são incluídos na distribuição oficial, pois não são código livre. O projeto Fedora Core somente distribui pacotes criados a partir de programas de código livre.

Para utilizar o repositório Livna, basta baixar e instalar o pacote RPM fornecido pelo repositório. Este pacote instala os arquivos para configurar o repositório junto ao Yum, além da chave pública usada para autenticar os pacotes, em um processo totalmente automatizado.

Todavia, nem sempre existem pacotes de chaves públicas disponíveis e, neste caso, será preciso importar manualmente uma chave pública de um empacotador para que rpm possa utilizá-la. Este é um processo relativamente fácil de ser feito. Para exemplificar como rpm lida com a checagem da integridade de pacotes, será mostrada a importação da chave do repositório **extras** do Fedora Core, á saber:

a) O pacote fedora-rpmdevtools-1.5-1.fc5.noarch.rpm foi obtido do repositório **extras**. Feita a checagem da integridade do pacote, é possível verificar que o sistema não possui a chave pública deste repositório. Assim, não é possível determinar se este é um pacote confiável.

```
$ rpm --checksig fedora-rpmdevtools-1.5-1.fc5.noarch.rpm
fedora-rpmdevtools-1.5-1.fc5.noarch.rpm: (SHA1) DSA sha1 md5 (GPG) NÃO-OK
(FALTAM AS CHAVES: GPG#1ac70ce6)
```

31 O repositório Livna pode ser encontrado em http://rpm.livna.org.

O próximo passo é importar a chave do repositório **extras** para que seja possível fazer a checagem de integridade e instalar o pacote com um grau de confiabilidade e segurança.

b) É necessário obter o arquivo da chave diretamente de uma fonte segura. Essa pode ser o *site* do repositório ou de um dos discos de instalação, pois é usual o empacotador distribuir a chave pública junto aos pacotes.

A instalação de uma chave pública é uma operação crítica para a segurança do sistema. Ao importar uma chave, o administrador afirma que confia totalmente na chave e nos pacotes assinados por essa chave pública. Caso uma chave falsa seja importada, pacotes adulterados poderão ser instalados e passarão pela checagem de segurança sem indicar qualquer falha.

No sistema Fedora Core, o arquivo com a chave do repositório **extra** está localizado no diretório `/etc/pki/rpm-gpg`. A importação de chaves para uso de rpm é uma operação exclusiva do superusuário `root`[32], pois se trata de uma operação administrativa que envolve aspectos de segurança do sistema.

```
# rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-fedora-extras
```

c) Realizada, novamente, a checagem, é possível confirmar o sucesso desta vez. A mensagem exibida: gpg Ok ; indica que o pacote é integro.

```
$ rpm --checksig fedora-rpmdevtools-1.5-1.fc5.noarch.rpm
fedora-rpmdevtools-1.5-1.fc5.noarch.rpm:(sha1) dsa sha1 md5 gpg OK
```

32 O usuário root é o administrador do sistema e possui os poderes totais para realizar funções administrativas.

4.3 – Administração das chaves públicas dos empacotadores.

De acordo com (Ewing; *et al*, 2002), após as chaves públicas serem importadas por rpm, é possível fazer a sua manutenção utilizando a sintaxe padrão para a manipulação de pacotes. Deste modo, uma consulta para obter informações sobre as chaves instaladas pode ser feita do seguinte modo:

```
# rpm -q gpg-pubkey-*
gpg-pubkey-1ac70ce6-41bebeef
```

Uma vez obtido o identificador (ID) da chave, é possível fazer uma consulta completa:

```
$ rpm -qi gpg-pubkey-1ac70ce6

Name        : gpg-pubkey          Relocations: (not relocatable)
Version     : 1ac70ce6            Vendor: (none)
Release     : 41bebeef            Build Date: Qui 27 Jul 2006 11:10:13 BRT
Install Date: Qui 27 Jul 2006 11:10:13 BRT      Build Host: localhost
Group       : Public Keys                   Source RPM: (none)
Size        : 0                                License: pubkey
Signature   : (none)
Summary     : gpg(Fedora Project <fedora-extras@fedoraproject.org>)
Description :

...Restante da saída da chave omitida...
```

A exclusão de uma chave pública também pode ser feita usando comandos de rpm, como no exemplo a seguir:

```
# rpm -e gpg-pubkey-1ac70ce6
```

5 – A base de dados RPM.

5.1 – Os arquivos da base de dados.

A base de dados RPM reside em `/var/lib/rpm`, sendo formada pelos arquivos descritos no Quadro 2.

Quadro 2 – Arquivos da Base de Dados RPM e suas Finalidades.

Arquivo	Finalidade
Basename	O arquivo base.
Conflictname	Armazena possíveis conflitos entre pacotes.
Dirnames	Armazena as rotas dos pacotes.
Filemd5s	Controle MD5 Sum dos arquivos.
Group	Controle dos Grupos de cada pacote.
Installtid	Controle ID de instalação.
Name	Nomes dos pacotes.
Packages	Controles de pacotes.
Providename	O que provê.
Provideversion	Qual versão provê.
Pubkeys	Chaves públicas dos desenvolvedores.
Removed	Controle de remoções.
Requirename	Controle dos requerimentos.
Requireversion	Controle das versões dos requerimentos.
Sha1header	Controla os sumários SHA1 do cabeçalho.
Sigmd5	Controla os sumários MD5 do cabeçalho.
Triggername	Controle de “gatilhos”.

Estes arquivos possuem, em sua maioria, o formato binário Berkeley DB. Segundo (Foster-Johnson, 2005), a biblioteca Berkeley DB pode ser descrita da seguinte maneira:

> A biblioteca Berkeley DB[33] fornece uma API simples para um banco de dados. Ela não é um banco de dados relacional tradicional. O mecanismo de armazenamento trabalha com uma tabela *hash* se referenciando aos pares de dados, como um nome/valor. Este tipo de banco de dados é muito rápido para verificar uma entrada (como o nome de um pacote) mas não é tão rápido para interagir com as outras entradas. Uma das coisas boas sobre esta biblioteca é que ela está disponível no modo de código aberto (*open-source*) e possui API para as linguagens C, C++, Java, Perl e Tcl. Cada um dos arquivos que formam a base de dados RPM possui uma função especializada para um tipo de consulta. Algumas vezes esta base de dados pode ser corrompida, sendo necessário reconstruir a base de dados.

5.2 – Reconstrução da base de dados.

Bancos de dados estão sujeitos à corrupção de um registro ou de uma tabela inteira. Nos casos mais extremos de corrupção da base de dados, pode ser necessário restaurar uma cópia de segurança. Todavia, na maioria dos casos, reindexar a base de dados é suficiente para restabelecer o funcionamento ou garantir a integridade do sistema. A opção `rebuilddb` promove a reindexação da base de dados à partir do arquivo base `/var/lib/rpm/Packages.` Os demais arquivos de índices serão recriados com as informações obtidas nesse arquivo. A sintaxe para realizar essa operação é a seguinte:

```
# rpm --rebuilddb
```

33 Disponível em www.sleepycat.com

Mesmo que o banco de dados esteja íntegro, ainda assim é possível executar esse comando para remover registros não usados reduzindo o tamanho dos arquivos de índice e otimizando as consultas à base de dados. É recomendável manter uma cópia de segurança antes de executar esse comando. Outra característica desse comando é que sua execução é demorada. Assim, é preciso ter paciência e aguardar que os índices sejam recriados. Um modo de verificar a integridade de uma base de dados é listar os pacotes e, em seguida, reconstruir a base de dados e comparar os resultados com uma nova consulta.

Caso as tentativas de recuperar uma base de dados tenham falhado, então, as informações sobre os pacotes instalados no sistema são perdidas. Neste caso, é possível iniciar um novo banco de dados à partir do zero.

Apesar da existência da possibilidade de continuar trabalhando com um sistema que teve sua base de dados perdida, essa não é uma atitude recomendada, pois afeta a segurança do sistema de arquivos. A opção, neste caso, é reinstalar o sistema e ser mais prevenido da segunda vez.

6 – Administração de sistemas GNU/Linux baseados em pacotes RPMs.

6.1 – Modos de operação de RPM.

O sistema RPM oferece sete modos básicos de operação para a manipulação de pacotes. O sistema RPM funciona como um banco de dados, desta maneira, os modos de operação podem ser vistos como métodos que permitem incluir, consultar, atualizar, verificar ou excluir os registros sobre os pacotes nessa base de dados. No Quadro 3, estão relacionados os modos de operação de RPM. É possível observar que cada modo de operação é acompanhado de um conjunto de parâmetros. Esses, situados entre colchetes, são opcionais e podem ser combinados entre si para obtenção de um resultado desejado. Os modos de operação e os parâmetros podem ser identificados por seu nome curto ou por um nome longo, como é usual na maioria dos comandos e parâmetros do *shell* do GNU/Linux.

Quadro 3 – Sintaxe dos Modos Básicos de Operação de rpm.

Operação	Função	Sintaxe
Query	Consulta	rpm {-q|--query} [select-options] [query-options]
Verify	Verifica	rpm {-V|--verify} [select-options] [verify-options]
Install	Instala	rpm {-i|--install} [install-options] PACKAGE_FILE
Upgrade	Atualiza	rpm {-U|--upgrade} [install-options] PACKAGE_FILE
Freshen	Restaura	rpm {-F|--freshen} [install-options] PACKAGE_FILE
Erase	Remove	rpm {-e|--erase} [--allmatches] [--nodeps] [--noscripts] [--notriggers][--repackage][--test]
Check	Segurança	rpm {-K|--checksig} [--nosignature] [--nodigest]

Segundo (Ewing; *et al*, 2002), além dos modos básicos de operação, existe um conjunto de parâmetros que são considerados opções gerais. Essas opções são usadas separadamente dos modos de operação, á saber:

-?, --help

Imprime uma longa lista de ajuda sobre vários parâmetros do comando.

--version

Imprime a versão do **rpm** em uso e encerra.

--quiet

Imprime o mínimo possível de mensagens, normalmente somente mensagens de erros serão exibidas.

-v

Exibe mais mensagens de informações.

-vv

Exibe várias informações úteis em um processo de depuração (*debug*).

--rcfile ***FILELIST***

Define uma relação de arquivos, separados por dois pontos, que realizam a configuração do aplicativo **rpm**. Os arquivos definidos em *FILELIST* são lidos seqüencialmente por **rpm**, sendo que somente é obrigatória a existência do primeiro arquivo. A esta lista, é acrescido o valor da variável **$HOME**. O valor padrão para *FILELIST* é (*/usr/lib/rpm/rpmrc*:*/usr/lib/rpm/redhat/rpmrc*: */etc/rpmrc* : *~/.rpmrc)*.

--pipe *CMD*

Direciona a saída de **rpm** para um comando definido em *CMD*.

--dbpath *DIRECTORY*

Define uma rota alternativa para os arquivos da base de dados. O valor padrão para *DIRECTORY* é */var/lib/rpm*.

--root *DIRECTORY*

O valor atribuído para *DIRECTORY* define a raiz do sistema de arquivos para as operações. Observe que isso significa que o caminho da base de dados será relativo ao valor definido em *DIRECTORY*. Este será usado para checagem das dependências ou por qualquer macro (por exemplo, por **%post** se instalando, ou **%prep** se construindo um pacote). Defina esse valor após um comando chroot(2) para *DIRECTORY.* Esses parâmetros são reunidos em quatro grupos de opções, conforme define (Ewing; *et al*, 2002):

Opções de Seleção (select-options)

[*PACKAGE_NAME*] [**-a,--all**] [**-f,--file** *FILE*] [**-g,--group** *GROUP*] {**-p,--package** *PACKAGE_FILE*] [**--fileid** *MD5*] [**--hdrid** *SHA1*] [**--pkgid** *MD5*] [**--tid** *TID*] [**--querybynumber** *HDRNUM*] [**--triggeredby** *PACKAGE_NAME*] [**--whatprovides** *CAPABILITY*] [**--whatrequires** *CAPABILITY*]

Opções de Consulta (query-options)

[**--changelog**] [**-c,--configfiles**] [**-d,--docfiles**] [**--dump**] [**--filesbypkg**] [**-i,--info**] [**--last**] [**-l,--list**] [**--provides**] [**--qf,--queryformat** *QUERYFMT*] [**-R,--requires**] [**--scripts**] [**-s,--state**] [**--triggers,--triggerscripts**]

Opções de Verificação (verify-options)

[**--nodeps**] [**--nofiles**] [**--noscripts**] [**--nodigest**] [**--nosignature**] [**--nolinkto**] [**--nomd5**] [**--nosize**] [**--nouser**] [**--nogroup**] [**--nomtime**] [**--nomode**] [**--nordev**]

Opções de Instalação (install-options)

[**--aid**] [**--allfiles**] [**--badreloc**] [**--excludepath** *OLDPATH*] [**--excludedocs**] [**--force**] [**-h,--hash**] [**--ignoresize**] [**--ignorearch**] [**--ignoreos**] [**--includedocs**] [**--justdb**] [**--nodeps**] [**--nodigest**] [**--nosignature**] [**--nosuggest**] [**--noorder**] [**--noscripts**] [**--notriggers**] [**--oldpackage**] [**--percent**] [**--prefix** *NEWPATH*] [**--relocate** *OLDPATH=NEWPATH*] [**--repackage**] [**--replacefiles**] [**--replacepkgs**] [**--test**]

6.2 – Modo de consultas.

O sistema RPM é uma base de dados que armazena informações sobre pacotes instalados em um sistema GNU/Linux. Deste modo, uma consulta (*query*) é uma operação realizada com certa freqüência por um administrador. O sistema de consultas de RPM possui uma flexibilidade comparável às instruções SQL[34] utilizadas para criar consultas personalizadas em bancos de dados relacionais. Todavia, como visto na seção `5.1`, como afirma (Foster-Johnson, 2005), "RPM usa o sistema de banco de dados Berkeley DB, que não é um sistema relacional, mas baseado em árvores binárias e pares de campos indexados por *hash*. O que torna o sistema muito rápido para esse tipo de consulta."

O sistema RPM não fornece relatórios padronizados de consulta, mas ferramentas, para que o administrador possa criar suas próprias consultas de acordo com suas necessidades pessoais. Esta flexibilidade é possível graças a uma série de rótulos (*tags*) que descrevem um campo de um registro de RPM.

Um rótulo é como uma variável que será substituída pelo conteúdo real do campo, armazenado em um registro. Desta maneira, é possível construir consultas personalizadas e com um nível de controle das informações bastante granular.

Além disso, também é viável o uso de comandos de consulta em *scripts* do *shell* ou inseridos no código de programas escritos com uma linguagem interpretada como `Perl` ou `Python`. Isso aumenta as possibilidades de uso das consultas RPM. A opção `querytags` relaciona os nomes válidos dos rótulos atualmente suportados por RPM:

```
# rpm --querytags
```

34 SQL - *Strutured Query Language* - Linguagem Estruturada de Consultas.

Além dos rótulos que permitem personalizar a saída das consultas, RPM oferece também uma série de comandos opcionais definidos no Quadro 3. Combinando os parâmetros das opções de seleção e de consulta é definida a sintaxe básica do modo de consulta descrita por (Ewing; *et al*, 2002) do seguinte modo:

```
rpm {-q|--query} [select-options] [query-options]
```

OPÇÕES DE SELEÇÃO POR PACOTE:

```
rpm {-q|--query} PACKAGE_NAME
```

Consulta se existe na base um pacote cujo nome é indicado por *PACKAGE_NAME*.

```
$ rpm -q bash
bash-3.1-6.2
```

-a, --all

Consulta o nome dos pacotes instalados em um sistema.

```
$ rpm -qa
curl-devel-7.15.1-1.2.1
libvorbis-1.1.2-1.2
rsync-2.6.6-2.2.1
gnome-menus-2.13.5-5.2
gtk2-engines-2.7.4-3
bash-3.1-6.2
... As demais saídas foram omitidas...
```

-f, --file *FILE*

Consulta qual o pacote instalou o arquivo especificado em *FILE*. Deste modo, é possível relacionar um arquivo com um pacote.

```
$ rpm -qf /bin/bash
bash-3.1-6.2
```

--fileid MD5

Consulta qual arquivo possui um determinado sumário MD5, retornando o nome do pacote que instalou o arquivo.

```
$ rpm -q --fileid 50b0fbd6e5f6c0754255ed25c83ae509
bash-3.1-6.2
```

-g, --group GROUP

Consulta os pacotes que pertencem a um determinado grupo. A relação completa dos grupos suportados no Fedora Core pode ser obtida em `/usr/share/doc/rpm-4.4.2/GROUP`.

```
$ rpm -q --group "System Environment/Shells"
tcsh-6.14-5.2.1
bash-3.1-6.2
```

--hdrid SHA1

Consulta qual pacote possui uma assinatura *SHA1*. Nota: Essa assinatura é imutável e faz parte do cabeçalho do pacote.

-p, --package PACKAGE_FILE

A flexibilidade de RPM permite também consultar informações sobre pacotes não instalados especificando seu nome em *PACKAGE_FILE*. O valor dessa variável pode apontar para um pacote que não seja local. Deste modo, os protocolos **FTP** ou **HTTP** são usados para baixar (*download*) o cabeçalho do pacote, que será lido para obtenção das informações.

--pkgid *MD5*

Consulta o pacote cujo identificador é *MD5*. O sumário MD5 é referente ao conteúdo (*payload*) e ao cabeçalho (*header*) do pacote.

--querybynumber *HDRNUM*

Consulta qual número do registro está associado ao pacote na base de dados. Essa opção é usada apenas para fins de depuração (*debugging*). Retorna o nome do pacote associado ao registro cujo número foi definido em *HDRNUM*.

```
$ rpm -q --querybynumber 45
xorg-x11-utils-1.0.1-1.2
```

--specfile SPECFILE

Consulta um arquivo *.spec* definido em *SPECFILE*, como se esse fosse um pacote. Entretanto, nem todas as informações (por exemplo, as listas de arquivos) estarão disponíveis para consulta.

--tid TID

Consulta os pacotes que possuem um determinado identificador de transação, definido em *TID*. Um rótulo com uma estampa de tempo no formato Unix (*Unix time stamp*) é correntemente usada como um identificador de transação. Os os pacotes instalados ou removidos em uma única transação possuem um identificador comum idêntico. O que não faz dessa uma boa opção de seleção.

--triggeredby PACKAGE_NAME

Consulta os pacotes que são "engatilhados" (*triggered)* por outros pacotes definidos em *PACKAGE_NAME*.

--whatprovides CAPABILITY

Consulta qual pacote fornece um recurso especificado em *CAPABILITY*.

```
$ rpm -q --whatprovides firefox
firefox-1.5.0.4-1.2.fc5
```

--whatrequires CAPABILITY

Consulta quais os pacotes requerem o pacote definido em *CAPABILITY* como dependência.

```
$ rpm -q --whatrequires yum
pirut-1.0.1-1
```

Os parâmetros das opções de consulta são assim definidos por (Ewing; *et al*, 2002) na documentação de RPM:

OPÇÕES DE CONSULTA POR PACOTE:

--changelog

Exibe o histórico de modificações do pacote. O histórico de modificações é uma lista formada pelas alterações que foram feitas em um programa para consertar erros de programação (*bugs*) ou para implementar mudanças no programa.

```
$ rpm -q --changelog coreutils

* Qua Mai 25 2005 Tim Waugh <twaugh@redhat.com> 5.2.1-48
- Prevent buffer overflow in who(1) (bug #158405).

* Sex Mai 20 2005 Tim Waugh <twaugh@redhat.com> 5.2.1-47
-  Better error checking in the pam patch (bug #158189).

-  ...As demais saídas foram omitidas...
```

-c, --configfiles

Lista os arquivos de configuração instalados pelo pacote.

```
$ rpm -q bash --configfiles
/etc/skel/.bash_logout
/etc/skel/.bash_profile
/etc/skel/.bashrc
```

-d, --docfiles

Lista os arquivos de documentação instalados pelo pacote. Os arquivos de documentação podem incluir páginas de manual `man` e documentos em outros formatos de arquivo como HTML, PDF e texto puro.

```
$ rpm -q coreutils --docfiles
...
/usr/share/man/man1/nice.1.gz
/usr/share/man/man1/nl.1.gz
/usr/share/man/man1/nohup.1.gz
/usr/share/man/man1/od.1.gz

... As demais saídas foram omitidas ...
```

--dump

Exibe diversas informações sobre o arquivo. O uso dessa opção é em conjunto com: **-l**, **-c**, **-d**. O cabeçalho dessa opção é assim definido:

```
caminho tamanho mtime md5sum modo dono grupo isconfig isdoc rdev ligação
```

```
$ rpm -q bash --dump

...As demais saídas foram omitidas...

/usr/share/man/man1/umask.1.gz 40 1139638495 4bdb94bc6cd57b2a9431dbfa590040f2
0100644 root root 0 1 0 X
/usr/share/man/man1/unalias.1.gz 40 1139638495 4bdb94bc6cd57b2a9431dbfa590040f2
0100644 root root 0 1 0 X
/usr/share/man/man1/unset.1.gz 40 1139638495 4bdb94bc6cd57b2a9431dbfa590040f2
0100644 root root 0 1 0 X
/usr/share/man/man1/wait.1.gz 40 1139638495 4bdb94bc6cd57b2a9431dbfa590040f2 0100644
root root 0 1 0 X
```

--filesbypkg

Lista o nome do pacote seguido do nome de um arquivo que pertence a esse pacote. Os os arquivos do pacote são listados.

```
$ rpm -q bash --filesbypkg
bash                     /bin/bash
bash                     /bin/sh
bash                     /etc/skel/.bash_logout
bash                     /etc/skel/.bash_profile
bash                     /etc/skel/.bashrc
bash                     /usr/bin/bashbug-32
```

-i, --info

Exibe informações sobre o pacote, incluindo o nome, a versão e uma descrição longa entre outras informações. As informações são retiradas da seção cabeçalho do pacote. Esta opção usa **--queryformat** se nenhum formato for especificado. Também é possível exibir qualquer uma das informações do cabeçalho separadamente usando rótulos.

```
$ rpm -qi bash

Name        : bash                         Relocations: /usr
Version     : 3.1                          Vendor: Red Hat, Inc.
Release     : 6.2                          Build Date: Sáb 11 Fev 2006 04:14:58 BRST
Install Date: Seg 17 Jul 2006 16:00:53 BRT   Build Host: hs20-bc1-6.build.redhat.com
Group       : System Environment/Shells     Source RPM: bash-3.1-6.2.src.rpm
Size        : 5298847                          License: GPL
Signature   : DSA/SHA1, Seg 06 Mar 2006 16:58:29 BRT, Key ID b44269d04f2a6fd2
Packager    : Red Hat, Inc. <http://bugzilla.redhat.com/bugzilla>
URL         : http://www.gnu.org/software/bash
Summary     : The GNU Bourne Again shell (bash) version 3.1.
Description :
The GNU Bourne Again shell (Bash) is a shell or command language interpreter that is
compatible with the Bourne shell (sh). Bash incorporates useful features from the
Korn shell (ksh) and the C shell (csh). Most sh scripts can be run by bash without
modification. This package (bash) contains bash version 3.1, which improves POSIX
compliance over previous versions.
```

--last

Ordena os pacotes em uma lista de acordo com a data e hora de instalação. Os pacotes mais recentes são exibidos no início da lista. Esta opção deve ser usada em

conjunto com um parâmetro para definir quais pacotes serão listados.

```
$ rpm -qg "System Environment/Shells"  --last

tcsh-6.14-5.2.1               Seg 17 Jul 2006 16:04:03 BRT
bash-3.1-6.2                  Seg 17 Jul 2006 16:00:53 BRT
```

-l, --list

Lista os arquivos e os diretórios instalados por um pacote.

```
$ rpm -ql bash

/bin/bash
/bin/sh
/etc/skel/.bash_logout
/etc/skel/.bash_profile
/etc/skel/.bashrc

...As demais saídas foram omitidas...
```

--provides

Lista as dependências que este pacote fornece.

```
rpm -q --provides kdelibs
config(kdelibs) = 6:3.5.1-2.3
libDCOP.so.4
libartskde.so.1
libkabc.so.1
libkabc_dir.so.1
libkabc_file.so.1
libkatepart.so
libkatepartinterfaces.so.0

...As demais saídas foram omitidas...
```

-R, --requires

Lista as dependências requeridas por um pacote.

```
$ rpm -q --requires bash

/bin/bash
/bin/sh
config(bash) = 3.1-6.2
libc.so.6
libc.so.6(GLIBC_2.0)
libc.so.6(GLIBC_2.1)
libc.so.6(GLIBC_2.2)
libc.so.6(GLIBC_2.3)
libc.so.6(GLIBC_2.3.4)
libc.so.6(GLIBC_2.4)
libdl.so.2
libdl.so.2(GLIBC_2.0)
libdl.so.2(GLIBC_2.1)
libtermcap.so.2
mktemp
rpmlib(CompressedFileNames) <= 3.0.4-1
rpmlib(PayloadFilesHavePrefix) <= 4.0-1
```

--scripts

Lista os *scripts* que são usados no processo de instalação ou desinstalação de um pacote.

```
$ rpm -q bash --scripts

... As demais saídas foram omitidas...

postuninstall scriptlet (using /bin/sh):
if [ "$1" = 0 ]; then
    grep -v '^/bin/bash$' < /etc/shells | \
        grep -v '^/bin/sh$' > /etc/shells.new
    mv /etc/shells.new /etc/shells
fi
```

-s, --state

Exibe o estado dos arquivos de um pacote. O estado de um arquivo pode ser apresentado como: normal, não instalado (*not installed)* ou substituído (*replaced).*

```
$ rpm -q bash --state
normal        /bin/bash
normal        /bin/sh
normal        /etc/skel/.bash_logout
normal        /etc/skel/.bash_profile
normal        /etc/skel/.bashrc
...As demais saídas foram omitidas...
```

--triggers, --triggerscripts

Exibe os *scripts* "gatilhos" (*triggers*), caso o pacote possua algum *script*.

```
$ rpm -q --triggers sendmail
triggerpostun scriptlet (using /bin/sh) -- sendmail < 8.10.0
/sbin/chkconfig --add sendmail
triggerpostun scriptlet (using /bin/sh) -- sendmail < 8.11.6-11
/usr/sbin/alternatives --auto mta
```

6.3 – Modo de verificação.

Os recursos de criptografia e assinaturas digitais, usados para proteger os pacotes da injeção de código malicioso, podem garantir a integridade dos pacotes antes de serem instalados. Entretanto, uma vez instalados no sistema, é preciso introduzir mecanismos que garantam a integridade desses arquivos ao longo do tempo. Uma forma de checar essa integridade é controlar certas propriedades de arquivos e diretórios. Assim, qualquer mudança em uma propriedade monitorada pode indicar problemas na segurança. Segundo (Uchôa, 2003), verificadores de integridade de arquivos como `AIDE` e `Tripwire` são as ferramentas indicadas para esse serviço.

Todavia, RPM também fornece um método de verificação que permite manter um estrito controle de certas propriedades dos arquivos instalados pelos

pacotes. De acordo com (Uchôa, 2003), a necessidade de manter esse controle surge a partir do seguinte histórico:

> Uma questão crítica no que se refere à segurança é a garantia de confiança no sistema. Em geral, tão logo o invasor obtém acesso ao sistema, sua primeira providência é garantir a continuidade desse acesso. Uma das estratégias utilizadas para isso é o uso de *rootkits*. Esses programas consistem em versões modificadas de aplicativos comuns ou do próprio *kernel*. Mesmo sem o uso de *rootkits*, pode ocorrer do invasor instalar um novo aplicativo que lhe dê acesso privilegiado.

RPM pode ajudar na tarefa de manter a integridade do sistema. Através das opções de verificação, é possível rastrear um conjunto de nove atributos dos arquivos instalados por RPM. Esses atributos são os seguintes:

- Proprietário;
- Grupo;
- Permissões;
- Assinatura MD5;
- Tamanho;
- Número maior e menor (apenas para *devices*[35]);
- Ligações simbólicas;
- Tempo de modificação.

Desta maneira, caso um invasor altere uma ou mais dessas propriedades, é possível ficar ciente dessas mudanças pela checagem de RPM. A checagem de RPM retorna, por padrão, apenas as indicações de inconsistências. Essa inconsistência é encontra pela comparação entre o valor de uma propriedade no sistema de arquivos e o valor correspondente armazenado na base RPM. O indicativo da inconsistência é uma cadeia de caracteres associados às propriedades. Assim, para cada arquivo verificado, cuja existência de falhas seja detectada, uma linha

35 *Devices* são um tipo especial de arquivo usados para definir dispositivos de *hardware* no diretório /dev.

com as propriedades modificadas será exibida.

O sintaxe dessas linha é: “[SM5DLUGT] x <*arquivo*>”; onde cada letra indica uma propriedade; conforme o Quadro 4, x é um campo que indica o tipo de arquivo para RPM e <*arquivo*> é o nome do arquivo verificado.

A sintaxe geral de um comando de verificação é:

```
rpm {-V|--verify} [select-options] [verify-options]
```

Quadro 4 – Caracteres retornados pela verificação de RPM.

Símbolo	Significado	Finalidade
S	Size (tamanho)	Monitora o tamanho do arquivo.
M	Mode (permissões)	Monitora as permissões.
5	Assinatura MD5	Monitora a assinatura MD5.
D	*Device*	Monitora os números maior/menor de um *device*.
L	Link (Ligação)	Monitora uma Ligação.
U	User (Dono)	Monitora o Dono.
G	Group (Grupo)	Monitora o Grupo.
T	Time Modification	Monitora o mtime.

Por exemplo, o pacote `bash` instala diversos manuais referentes a seus utilitários. A verificação de RPM pode ser demonstrada pela introdução de modificações controladas em dois arquivos de documentação `man` desse pacote.

Os manuais estão armazenados no diretório em `/usr/share/man/man1`. Um arquivo será renomeado e o outro terá alterada a propriedade de dono e de grupo. Em seguida, será feita a verificação desse pacote por RPM.

Os comandos a seguir realizam as modificações nos dois arquivos:

```
# cd /usr/share/man/man1
# mv times.1.gz copia.times.1.gz
# chown unasi.unasi umask.1.gz
```

A verificação de RPM acusa a ausência (*missing*) do arquivo renomeado e mostra os caracteres (UG) indicando que o dono (*User*) e o grupo (*Group*) do outro arquivo foram alterados. Um campo verificado e que não apresentou falhas de segurança é indicado por um ponto. Saídas literais como *missing* indicam o estado atual do pacote. Neste caso, a sua ausência, uma vez que o arquivo foi renomeado, não pôde ser encontrado por RPM. A apresentação de um sinal de interrogação (?) significa que este campo não pôde ser verificado. Normalmente, a causa disso é a insuficiência de permissões do usuário que realiza o teste. Deste modo, RPM não tem como determinar essa propriedade, pois depende da permissão para realizar o teste, ficando o valor indefinido.

```
# rpm -V bash
missing   d /usr/share/man/man1/times.1.gz
..?..UG.. d /usr/share/man/man1/umask.1.gz
```

De acordo com (Ewing; *et al*, 2002), durante uma verificação padrão, todos os atributos são verificados. Entretanto, é possível desabilitar a verificação de alguns atributos do processo através do uso das opções de verificação (*verify-options*), a seguir relacionadas:

```
[--nodeps] [--nofiles] [--noscripts] [--nodigest] [--nosignature] [--nolinkto] [--nomd5] [--nosize] [--nouser] [--nogroup] [--nomtime] [--nomode] [--nordev]
```

6.4 – Modos de instalação, atualização e restauração.

Mais cedo ou mais tarde o administrador de um sistema GNU/Linux precisará modificar seu sistema através de uma operação de atualização ou de instalação. Um dos principais motivos que levam a realizar uma dessas operações são as correções de segurança. Outro importante motivo é a evolução do código dos programas que formam um pacote. Os desenvolvedores podem introduzir funcionalidades que um administrador pode precisar em seus sistemas.

Seja para atualizar (*upgrade*), restaurar (*freshen*) ou instalar (*install*) novos pacotes, o administrador precisará conhecer as pequenas diferenças entre esses modos de operação antes de utilizá-los. Assim, essas três operações, apesar de conceitualmente diferentes, podem ser vistas em um único tópico, pois, na prática, são bastante similares. As diferenças de cada caso vão definir realmente qual dessas operações utilizar:

- O modo de instalação (`--install`) é usado quando um administrador deseja instalar um novo pacote no sistema. Desta maneira, não havendo qualquer versão anterior deste pacote instalada no sistema, essa é a opção certa para usar.
- O modo de atualização (`--upgrade`) é usado quando o quê um administrador deseja é substituir um pacote instalado por uma versão mais atual deste mesmo pacote. Existindo uma versão anterior, ela será removida do sistema para dar lugar à nova versão. No caso de não existir uma versão anterior instalada, o sistema RPM fará a instalação do pacote como uma nova instalação. Desta maneira, no modo de instalação, o pacote é instalado e não há preocupação com versões anteriores. RPM permite a instalação de várias versões diferentes do mesmo pacote, mesmo que a base de dados fique inconsistente. No modo de atualização, RPM removerá a versão anterior. Usar a opção de instalação no lugar da opção de

atualização, na maioria dos casos, torna a base de dados RPM inconsistente.

- O modo de restauração (`--freshen`) é utilizado quando é necessário atualizar um pacote caso. Isso significa que, se não existir uma versão anterior deste programa instalada, então, o sistema encerra sem instalar qualquer arquivo. Essa é a diferença básica entre os modos de atualização e de restauração. O significado da palavra restauração, neste contexto, tem o sentido de "deixar novo" ou "refrescar". Neste sentido, se aproxima mais do termo atualização. Assim, enquanto o modo de instalação instala, o modo de atualização atualiza ou instala, o modo de restauração somente atualiza um pacote. Apesar das diferenças sutis de cada caso é preciso estar atento para qual opção usar.

De acordo com (Foster-Johnson, 2005), tanto a opção -U ou -i podem ser usadas para instalar um pacote. Ambas realizam uma série de passos que permitem preparar o sistema para a instalação, instalar e realizar ajustes após uma instalação. Esses passos são descritos do seguinte modo:

- Checagem das dependências necessárias para o pacote funcionar;
- Checagem por eventuais pacotes que possam representar conflitos;
- Realização de tarefas de pré-instalação, como a criação de diretórios;
- Descompressão dos arquivos do pacote, que se encontram no formato Gzip, diretamente em seus diretórios de instalação;
- Realização de tarefas de pós-instalação, como a remoção de arquivos temporários usados no processo;
- Atualização da base de dados RPM. Esta operação mantém a integridade da base de dados.

De acordo com (Ewing; *et al*, 2002), a sintaxe básica para cada um desses modos é a seguinte:

```
rpm {-i|--install} [install-options] PACKAGE_FILE
rpm {-U|--upgrade} [install-options] PACKAGE_FILE
rpm {-F|--freshen} [install-options] PACKAGE_FILE
```

Os parâmetros das opções de instalação (*install-options*) são compartilhados pelos três modos e podem ser utilizados de acordo com suas finalidades. Desta maneira, é preciso estar atento para qual opção usar adequadamente.

De acordo com (Ewing; *et al*, 2002), a sintaxe dos parâmetros de instalação podem ser assim descritas:

--allfiles

Instala ou atualiza os arquivos de um pacote, ignorando se eles existem.

--badreloc

Usada com a opção **--relocate**, permite realocar os caminhos (*paths*), não apenas aquelas constantes em *OLDPATH* e incluídos na sugestão do pacote binário.

--excludepath *OLDPATH*

Não instala arquivos cujo caminho comece com *OLDPATH*.

--excludedocs

Não instala qualquer arquivo que esteja marcado como documentação (isso inclui páginas de manual e documentos texinfo).

--force

Equivale a ***--replacepkgs, --replacefiles,*** e ***--oldpackage.***

-h, --hash

Imprime uma barra indicativa (50 vezes o caractere #) demonstrando visualmente o andamento do processo de instalação de um pacote. Usada com a opção **-v**.

--ignoresize

Não checa antes se o espaço no sistema de arquivos é suficiente para instalar o pacote.

--ignorearch

Permite que a instalação ou atualização de um pacote mesmo que a arquitetura para o qual o pacote foi construído não combine com a arquitetura do *host*.

--ignoreos

Permite que a instalação ou atualização de um pacote mesmo que o sistema operacional para o qual o pacote foi construído não combine com o S.O do *host*.

--includedocs

Instala os arquivos de documentação. Este é o comportamento padrão.

--justdb

Atualiza somente a base de dados, mas não atualiza o sistema de arquivos.

--nodigest

Não verifica o sumário do pacote ou do cabeçalho quando lendo o pacote.

--nosignature

Não verifica a assinatura do pacote ou do cabeçalho quando lendo o pacote.

--nodeps

Não faz a checagem das dependências antes de instalar ou atualizar um pacote.

--nosuggest

Não exiba sugestões de pacotes para prover uma dependência que esteja faltando.

--noorder

Não reordena os pacotes para uma instalação. Uma lista de pacotes pode será normalmente reordenada para satisfazer dependências.

--noscripts

--nopre

--nopost

--nopreun

--nopostun

Não executa a macro de mesmo nome. A opção `--noscripts` é equivalente ao

uso de todas juntas e desativa a execução das seguintes macros: `%pre`, `%post`, `%preun`, e `%postun`.

`--notriggers`

`--notriggerin`

`--notriggerun`

`--notriggerpostun`

Não executa nenhuma macro "gatilho" (*trigger*) de mesmo nome. A opção --notriggers é equivalente ao uso de todas juntas e desativa a execução das seguintes macros:

`%triggerin, %triggerun, e %triggerpostun`

`--oldpackage`

Permite que uma operação de atualização substitua um pacote atualmente instalado por outro cuja versão seja mais antiga. Normalmente, RPM não permite a operação de rebaixamento de versão (*downgrade*).

`--percent`

Exibe as porcentagens à medida que os arquivos são descompactados. Esta função pode ser usada quando outros programas ou ferramentas que executam **rpm**.

`--prefix` ***NEWPATH***

Redefine a localização para instalação dos arquivos de um pacote, direcionando os arquivos para o caminho definido na opção *NEWPATH*.

***`--relocate`* OLDPATH=NEWPATH**

Permite realocar pacotes binários, traduzindo os caminhos que se iniciam com *OLDPATH* na sugestão de relocação do pacote, para o novo valor em *OLDPATH*. Esta opção pode ser usada repetidamente se diversas variáveis *OLDPATH* serão realocadas.

--repackage

Reempacota os arquivos novamente em um pacote antes de apagá-los. O pacote previamente instalado será nomeado de acordo com o valor da macro `%_repackage_name_fmt` e será criado no diretório indicado pela macro `%_repackage_dir` (O valor padrão é */var/spool/repackage*).

--replacefiles

Instala os pacotes mesmo que ele substitua arquivos de pacotes já instalados.

--replacepkgs

Instala os pacotes se alguns deles estiverem instalados anteriormente.

--test

Não instala o pacote, simplesmente checa e avisa sobre potenciais conflitos.

Os modos de instalação, atualização ou restauração são funcionalmente idênticos. Os exemplos de uso a seguir descrevem alguns casos de uso:

a) Teste da instalação de um pacote que não tem uma versão anterior instalada no sistema.

```
# rpm -ivh --test coreutils-5.96-1.2.i386.rpm
A preparar...
########################################### [100%]
```

b) Atualização de um pacote, seguido da operação de reempacotamento dos arquivos removidos em um pacote no diretório */var/spool/repackage.*

```
#  rpm -Uvh --repackage  util-linux-2.13-0.20.1.i386.rpm
A preparar...           ########################################### [100%]
Repackaging...
   1:util-linux         ########################################### [100%]
Upgrading...
   1:util-linux         ########################################### [100%]
```

c) Atualização de um pacote instalado.

```
# rpm -Fvh totem-1.4.1-1.i386.rpm
A preparar...
########################################### [100%]
   1:totem
########################################### [100%]
```

6.5 - Modo de exclusão.

Um dos grandes benefícios introduzidos por um sistema de gerenciamento de pacotes como RPM é a facilidade com que se instala um grande número de arquivos através de um único pacote e tudo com apenas uma linha de comando. Entretanto, além das modificações por instalações e atualizações de pacotes, pode chegar o momento em que um ou mais pacotes não sejam mais necessários. Neste caso, a solução é a remoção destes pacotes do sistema.

Pensando nisso, os projetistas de RPM implantaram as mesmas facilidades encontradas no modo de instalação também no modo de remoção de pacotes.

Apesar da remoção de pacotes não ser uma tarefa muito comum em sistemas servidores, em ambientes de testes ou em uma estação de trabalho para uso pessoal, a remoção de pacotes é rotineira. É comum, nestes casos, experimentar um programa e desistir de utilizá-lo pouco tempo depois.

Para um usuário comum, pode não ser tão visível a vantagem de ter um sistema como RPM controlando a remoção de programas do sistema. O usuário leigo pode pensar que o mais simples é remover o diretório de instalação e está resolvida a questão da exclusão de um programa.

Mas, não é tão simples assim. A remoção de um conjunto de arquivos pode fazer com que outros programas também parem de funcionar. Os sistemas

GNU/Linux são fortemente baseados em compartilhamento de rotinas através de arquivos de bibliotecas de funções.

Entretanto, se, por um lado, um dos grandes trunfos introduzidos pelo sistema de gerenciamento de pacotes RPM é a checagem das dependências durante a instalação, por outro, também é importante a segurança trazida pelo sistema durante a remoção de um programa. Através de estrito controle das relações entre os pacotes, RPM pode detectar e impedir a remoção de qualquer pacote que seja requerido por outro para funcionar corretamente. Deste modo, o sistema impede que a instalação se torne inconsistente por causa da remoção indevida de um pacote.

O processo de remoção é um modo simples para eliminar qualquer vestígio de um pacote instalado em um sistema. Ainda existem as opções de pré e pós-remoção. Essas opções permitem manter o controle do que deve ser feito antes, durante e depois que um pacote é removido. Como demonstra (Foster-Johnson, 2005), "as macros `%preun` e `%postun` são executadas antes e após um processo de desinstalação ajustando o ambiente enquanto um pacote é preparado para ser removido ou após essa remoção".

De acordo com (Bailey, 2000), uma série de passos são realizados para que um pacote possa ser removido do sistema com um único comando:

- Uma checagem na base de dados confirmando as dependências;
- É executado um roteiro (*script)* de pré remoção. Caso exista um;
- É feita uma checagem para verificar se os arquivos de configuração do pacote foram alterados. Caso positivo, é feita uma cópia de segurança desses arquivos com seu nome alterado;
- É checado se cada arquivo listado como parte deste pacote é exclusivo do pacote ou compartilhado com outros pacotes. Somente no caso em que seja exclusivo, então, o arquivo é removido;

- É executado um roteiro (*script*) de pós remoção. Caso exista um;
- Remoção de todos os registros referentes ao pacote da base de dados.

Conforme relaciona (Ewing; *et al*, 2002), o formato geral de um comando de remoção usando rpm é:

```
rpm {-e|--erase} [--opções] PACKAGE_NAME
```

As seguintes opções podem ser usadas:

--allmatches

Remove as versões do pacote que combinam com a variável *PACKAGE_NAME*. Se vários pacotes combinam com o valor desta variável e for tentada a remoção sem usar essa opção, então uma mensagem de erro é exibida.

```
# rpm -e --allmatch gpg-pubkey-*
```

--nodeps

Não checa as dependências antes de desinstalar um pacote.

--noscripts

--nopreun

--nopostun

Não executa as macros de mesmo nome.

A opção **--noscripts** usada durante a remoção de pacotes é equivalente a:

--nopreun --nopostun

e desativa a execução das seguintes macros: **%preun, e %postun**

--notriggers

--notriggerun

--notriggerpostun

Não executa qualquer macro "gatilho" (*trigger*) do tipo determinado. A opção**--notriggers** é equivalente a: **--notriggerun --notriggerpostun** e desativa a execução das macros: **%triggerun, e %triggerpostun**

--repackage

Reempacota os arquivos antes de apagá-los. O pacote previamente instalado será renomeado de acordo como o valor da macro **%_repackage_name_fmt** e será criado no diretório definido pela macro **%_repackage_dir**, o valor padrão para este parâmetro é */var/spool/repackage*.

Esta opção reconstrói um pacote antes de removê-lo. Pacotes assim não são completos e não devem ser instalados novamente.

--test

Não desinstala realmente um pacote, apenas simula a remoção. Usada juntamente com a opção **-vv** para fins de depuração.

Entre os parâmetros possíveis de serem usados no processo de remoção, talvez o mais interessante seja o modo de teste. Neste caso, é possível testar como o sistema procederia a remoção apenas de modo simulado. Nada é realmente removido.

```
# rpm -evv --teste eject
```

7 -Desenvolvimento de pacotes RPM

7.1 - Introdução a criação de pacotes RPM.

Neste ponto é possível ter uma visão dos amplos benefícios de um sistema de empacotamento de programas como RPM. As tarefas de instalação, atualização e remoção são reduzidas literalmente a uma linha de comando. O sistema RPM torna uma tarefa complexa como a instalação de programas em sistemas GNU/Linux algo simples. Para entender esse e outros pontos do funcionamento de RPM, é preciso entender a sua concepção.

Dois pilares do desenvolvimento de RPM o tornam o sistema robusto e transparente: filosofia e programação. O sistema RPM é amigável para o usuário final e prático para desenvolvedores graças a uma filosofia de desenvolvimento e a uma programação baseada nesses princípios. Segundo (Bailey, 2000), a filosofia de desenvolvimento de RPM teve como uma de suas linhas básicas tornar mais fácil a vida de usuários e, principalmente, de desenvolvedores de pacotes. O mesmo é visto nas afirmações de (Foster-Johnson, 2005):

> Os desenvolvedores de RPM, especialmente Marc Ewing e Erik Troan, baseados em suas experiências anteriores com empacotamento de programas Linux e seus conhecimentos sobre ferramentas de empacotamento de outras plataformas, tinham em mente facilitar a vida de desenvolvedores que utilizassem o sistema RPM. Para isso, elegeram uma série de pontos e características que deveriam estar sempre presentes no desenvolvimento de RPM:

- Facilidade para o uso;
- Foco orientado para o pacote;
- Tornar um pacote atualizável;
- Rastreamento das interdependências dos pacotes;
- Capacidade de consulta a uma base de dados;
- Segurança por verificação;

- Suporte para múltiplas arquiteturas;
- Preservação do código-fonte original (*Pristine Source*).

Enquanto a implementação de alguns destes conceitos é percebida nos modos de instalação, atualização e remoção, outras destas características são visíveis somente para desenvolvedores de pacotes. Desta maneira, antes de partir para o desenvolvimento de pacotes RPM, é preciso antever alguns destes conceitos relacionados com o desenvolvimento de pacotes.

Na visão de (Bailey, 2000), “um desenvolvedor pode decidir distribuir seu próprio programa no formato RPM, todavia, e se acredita que na maioria dos casos, um empacotador está distribuindo um programa que não foi desenvolvido por ele.” Como tradicionalmente as aplicações GNU/Linux são programas de código aberto, é comum distribuidores também serem desenvolvedores.

Deste modo, esses distribuidores acabam por implementar mudanças no código, seja para adaptar melhor os programas ao seu sistema ou mesmo para corrigir erros de programação no código original. Estas sucessivas mudanças levavam a uma situação de perda de controle sobre as mudanças e acabavam por descaracterizar o código-fonte original. Assim, não era mais possível separar o que é código original do que é atualização. Para (Bailey, 2000), isso levou à seguinte solução:

> É raro uma aplicação obtida na Internet não ter seu código-fonte alterado antes de ser empacotado. Neste ponto, entra o conceito de código original (*Pristine Source*). Quando este código é empacotado por RPM, ele é integralmente incluído no pacote, mantendo os arquivos que formam o programa sem qualquer modificação. As atualizações (*patches*) são mantidas separadas do código original, em uma coleção de atualizações.

Deste modo, as atualizações (*patches*) são mantidas separadamente em relação ao código-fonte original, permitindo rastrear as múltiplas versões de um

pacote. Deste modo, um desenvolvedor passa a possuir um arquivo com o código-fonte original, normalmente um arquivo do tipo `tar.gz`, um conjunto de atualizações (*patches*) que modifica esse código-fonte e arquivos de suporte (caso sejam necessários) como os *scripts* do tipo *System V Init* ou outros. Juntas, estas partes formarão um pacote fonte RPM que dará origem aos pacotes binários RPM.

7.2 - O programa-exemplo

Para ilustrar o processo de construção de um pacote RPM e os conceitos envolvidos nessa tarefa, é preciso ter um programa-exemplo que sirva de referência na demonstração prática deste trabalho. Para tanto, utilizar um programa real poderia tornar o exemplo muito complexo.

A solução foi criar um pequeno programa cuja função é servir de protótipo para a criação do pacote. O programa-exemplo será chamado ARL. Este programa foi escrito utilizando a linguagem C padrão e possui um único arquivo de código-fonte denominado "`arl.c`", que será usado para gerar um arquivo objeto "`arl.o`" que após o processo de ligação, feito pelo compilador, dará origem a um executável chamado `arl`. A finalidade deste programa-exemplo é exibir o texto "ARL- Administração de Redes Linux:" seguido da data e da hora do sistema quando invocado.

Propositalmente,o programa ARL possui um pequeno ajuste em seu código de programação. Neste caso, é preciso criar uma atualização (*patch*) para que este programa funcione como desejado. O código-fonte do arquivo "`arl.c`" pode ser integralmente visto na listagem a seguir:

```
# include <stdio.h>
# include <time.h>

int main ()
{
struct tm *ptr;
time_t lt;
lt = time(NULL);
ptr  = localtime(&lt);
printf ("ARL- Administração de Redes Linux: %s \n", asctime(ptr));
}
```

7.3 - Criação de uma atualização

De acordo com (Guru Labs, 2005), "quando um programa está sendo preparado para empacotamento, normalmente ele precisa de modificações em seu código-fonte. São vários os motivos para fazer modificações, sendo o mais comum aplicar correções ao código para eliminar erros de programação."

Uma atualização (*patch*) é um trecho de código-fonte que reflete uma mudança no código original de um programa. Segundo (Guru Labs, 2005), o processo de criação de uma atualização pode ser assim descrito:

> Atualizações são facilmente criadas. Para criar uma atualização (*patch*), basta obter o código-fonte original e fazer uma cópia deste arquivo incluindo a extensão de arquivo *`.orig.`* Em seguida, basta fazer as alterações necessárias no código. O próximo passo é usar o comando `diff`[36] para criar um arquivo com as diferenças entre o arquivo alterado e o original. Esse arquivo com as diferenças é um *patch* a ser aplicado na criação de um pacote.

Supondo que a aplicação-exemplo denominada `arl` versão 2004.2 será empacotada no formato RPM os procedimentos para tal tarefa podem são descritos a seguir:

36 O utilitário `diff` faz parte do pacote `diffutils.` É usado para comparar as diferenças entre dois arquivos.

O arquivo de código-fonte é denominado arl-2004.2.tar.gz e, ao ser desempacotado pelo comando tar, dá origem a um diretório de nome arl-2004.2. Esse programa-exemplo possui um arquivo de código-fonte, escrito na linguagem C, denominado arl.c. , que será modificado. É preciso criar uma atualização para esse programa, de modo que o ajuste na programação seja feito antes que o programa possa ser empacotado.

No intuito de criar uma atualização (*patch*) para este programa, os seguintes procedimentos devem ser feitos:

- Descompactar o código-fonte;
- Criar uma cópia de arl.c com o nome de arl.c.orig;
- Editar arl.c com um editor de texto e corrigir o erro. Em seguida, salvar as mudanças;
- Gerar um arquivo com as diferenças entre os programas originais e os programas modificados. Para isso usar o comando diff.

```
# diff -Naur arl.c.orig arl.c > arl.fix1.patch
```

Esse arquivo com as diferenças (arl.fix1.patch) será incluído como uma atualização no momento da criação do pacote, como será demonstrado mais adiante neste trabalho. É possível visualizar a linha onde houve a correção do programa. Neste exemplo, um trecho de código (\n) foi removido. Uma listagem do conteúdo desse arquivo de atualização pode ser vista a seguir:

```
--- BUILD/arl-2004.2/arl.c.orig 2006-08-10 16:02:48.000000000 -0300
+++ BUILD/arl-2004.2/arl.c      2006-08-10 16:03:01.000000000 -0300
@@ -7,5 +7,5 @@
 time_t lt;
 lt = time(NULL);
 ptr  = localtime(&lt);
-printf ("ARL- Administração de Redes Linux: %s \n", asctime(ptr));
+printf ("ARL- Administração de Redes Linux: %s", asctime(ptr));
 }
```

7.4 – Conceitos sobre o arquivo *.spec*

Segundo (Bailey, 2000), o arquivo *.spec* é usado por RPM para controlar o processo de construção de um pacote. Um arquivo deste tipo é uma coleção de instruções formada por macros usadas por RPM, podendo ou não possuir instruções de comandos do *shell* ou outra linguagem interpretada. Podem ser usadas Perl ou Python, por exemplo.

Esse conjunto de macros RPM e comandos interpretados vai guiar o processo de compilação do código-fonte, dando origem a um aplicativo. Posteriormente, RPM pode criar tanto um pacote binário quanto um pacote fonte ou ambos. Para tanto, utilizará os arquivos gerados nesse processo de construção.

O arquivo *.spec* é o ponto central da produção de um pacote. A qualidade da construção de um arquivo *.spec* determina todo o processo de criação do pacote. De acordo com (Bailey, 2000), "um arquivo *.spec* tem muitas responsabilidades e tarefas durante a construção de um pacote, o que acaba por torná-lo um pouco complexo. O modo encontrado para diminuir esta complexidade foi a sua divisão em seções". Cada uma delas fica responsável por uma das partes relacionadas com o processo de criação de um pacote, como se pode observar analisando cada uma dessas seções.

7.4.1 - Seção preâmbulo.

Segundo (Bailey, 2000), "a primeira seção de um arquivo *.spec* é o preâmbulo, cuja função é armazenar os dados sobre o pacote. Estes serão exibidos quando um usuário consultar informações sobre o pacote." O preâmbulo é formado por rótulos, um por linha, seguidos por dois pontos e uma atribuição de valor para o rótulo. Como no exemplo a seguir:

```
Summary: ARL - Administração em Redes Linux
Name: arl
Version: 2004.2
Release: 1
License: GPL
Group: "Pós-Graduação/Lato Sensu"
URL: http://www.ginux.comp.ufla.br/arl
Vendor: UFLA/FAEPE
Packager: Keynes Augusto <unasi@nospam.com>
Source: %{name}-%{version}.tar.gz
Patch0: arl.fix1.patch
Prefix: /usr/local/bin
%description
Este é o pacote RPM para o famoso ARL. ARL é um comando usado para
exibir a data e a hora local de um sistema GNU/Linux.
```

Os rótulos `name`,`version` e `release` armazenam, respectivamente, o nome, o número da versão e da atualização do pacote. Os valores incluídos nestes rótulos serão usados pela base de dados de RPM para referenciar o pacote. O nome do pacote normalmente reflete o nome do aplicativo que ele implementa. Mas, isso não é uma regra rígida, podem haver exceções.

Os valores armazenados nestes rótulos não têm qualquer ligação real com o nome do arquivo RPM. Caso um arquivo RPM seja renomeado, ainda assim poderá ser instalado. As informações destes rótulos são mantidas inalteradas quando ocorre uma renomeação de arquivo. Um nome de arquivo é somente uma convenção. O que define o tipo de um arquivo é o seu número mágico.

O rótulo `License` indica o tipo de licenciamento do aplicativo. O rótulo `Group` indica a qual categoria o aplicativo pertence. O rótulo `URL` aponta para um endereço na Internet onde é possível encontrar informações sobre o programa. Normalmente, este é o endereço dos desenvolvedores desta aplicação. O rótulo `Packager` armazena o nome e o contato do empacotador do programa. O contato pode ser um endereço de correio eletrônico ou a URL de um *site* de controle de erros, como, por exemplo, `<http://bugzilla.redhat.com/bugzilla>`.

O rótulo `Source`, neste exemplo, faz uso dos rótulos `RPM`: `%{name}` e

%{version}, empregando esses rótulos como variáveis. Os mesmos serão substituídos por seus valores definidos anteriormente no próprio arquivo *.spec.* É possível utilizar qualquer rótulo válido e reconhecido por RPM.

O rótulo Patch é utilizado para listar as atualizações que serão aplicadas ao código-fonte original. As atualizações devem ser incluídas na ordem que serão aplicadas e os respectivos arquivos devem estar salvos no diretório SOURCES, criado para abrigar os fontes durante a geração de um pacote. Os rótulos BuildRequires e Requires são opcionais, pois RPM calcula as dependências em tempo de execução. Entretanto, é desejável que as dependências e requerimentos sejam relacionadas, já que isso facilita a compilação do pacote. Por fim, o rótulo %description é usado para criar um pequeno texto com a descrição sobre a funcionalidade que o pacote implementa.

7.4.2 - Seção de preparação - (%prep)

Para (Bailey, 2000), "enquanto no preâmbulo a maioria das informações é para consumo humano e têm pouca relação com o processo de construção do pacote, na seção %prep o foco é inteiramente voltado para os processos que preparam o programa para ser construído". Deste modo, é nessa seção que são incluídas as instruções para proceder as seguintes tarefas:

- Criação de um diretório onde a aplicação será construída;
- Descompactação do código-fonte dentro deste diretório;
- Aplicação de *patches*, se necessário;
- Realização de qualquer operação necessária para deixar os fontes prontos para a compilação.

À primeira vista, essa seção parece com um *script* do *shell*. Todavia, essa impressão não é errada, pois é isso mesmo que ela é. Essa seção é tida como uma das mais complexas de um arquivo *.spec*. Contudo, RPM fornece objetos pré-programados que facilitam bastante sua construção. Esses objetos recebem o nome de macros RPM. De acordo com (Guru Labs, 2005), as macros podem ser compreendidas do seguinte modo:

> Macros são amplamente usadas para realizar configurações em variadas partes de RPM. Essas configurações podem ser globais ou pessoais. As macros globais são armazenadas em `/usr/lib/rpm/macros`. As macros de uso pessoal são armazenadas no arquivo `.rpmmacro` que cada usuário, construtor de pacotes, possui em seu diretório pessoal. As macros definidas na configuração pessoal vão se sobrepor às macros globais. As macros são desenvolvidas pelos distribuidores e apresentam diferenças entre si. É recomendável cuidado no uso de macros quando um pacote for para uso de mais de uma distribuição.

Na seção `%prep`, existem duas macros que facilitam o trabalho de desenvolvimento de um pacote: as macros `%setup` e `%patch`. A primeira é utilizada para criar a estrutura de diretórios necessária para a compilação e para descompactar o código-fonte nestes diretórios, enquanto a segunda é usada na aplicação de atualizações (*patches)*, quando isso for necessário.

Como citado anteriormente, é nessa seção que o código-fonte deve ser totalmente preparado para compilação. Isso inclui descompactação dos fontes, aplicação dos *patches* e checagem dos requisitos para compilação. As macros `%setup e %patch` fazem as duas primeiras funções, mas para realizar a checagem dos requisitos é preciso usar as ferramentas denominadas `autotools`.

Segundo (Camargo, 2005), "o projeto GNU desenvolve ferramentas que fazem a checagem dos requisitos do sistema e buscam facilitar o processo de compilação de um programa. Ferramentas como: `autoconf, automake, autoheader e libtool`, são chamadas genericamente de `autotools`". Essas ferramentas são usadas para criar os "conhecidos" *scripts* `configure e Makefile`,

geralmente utilizados durante a compilação de programas no GNU/Linux.

No capítulo "Ferramentas de Desenvolvimento", em (Camargo, 2005), é feita uma introdução sobre o assunto. Não são explicados os detalhes do funcionamento destas ferramentas, pois isso está fora do escopo deste trabalho. Para mais detalhes, consulte (Camargo, 2005). Esses arquivos automatizarão o processo de configuração e compilação do programa-exemplo `arl`. A seguir, são listados os arquivos `configure.in` e `Makefile.in`, usados para exemplificar a construção do pacote ARL.

Arquivo configure.in:

```
AC_INIT(arl.c)
AC_PROG_CXX
AC_LANG_C
AC_PROG_MAKE_SET
AC_CHECK_LIB(ncurses,main,,AC_MSG_ERROR(Instale ncurses))
AC_OUTPUT(Makefile)
```

Arquivo Makefile.in:

```
CXX = @CXX@
CFLAGS = @CXXFLAGS@
LDFLAGS = @LDFLAGS@

arl: arl.o
        $(CXX) $(LDFLAGS) $< -o $@
arl.o: arl.c
        $(CXX) $(CXXFLAGS) -c $<
clean:
        $(RM) arl.o

distclean:
        $(RM) arl config.* *.o Makefile
all: arl
```

A função do arquivo `configure.in` é checar se o ambiente possui os requisitos básicos necessários para a compilação. É checada a presença de compiladores, bibliotecas e dependências. Caso os requisitos estejam resolvidos, en-

tão é gerado um arquivo Makefile.

Este, por sua vez, somente será usado na seção %build, responsável pela compilação do código-fonte e geração dos executáveis. A seguir, é apresentado um exemplo da seção %prep que faz uso das macros %setup e %patch.

```
# Seção PREP
%prep
%setup
cd %{_topdir}
%patch -P 0
cd $RPM_BUILD_DIR/%{name}-%{version}
./configure
```

7.4.3 - Seção de compilação (%build).

Uma vez que o código-fonte está ajustado, é chegado o momento da compilação. Essa tarefa é realizada na seção %build. Essa seção não possui macros, somente são usados comandos de terminal relacionados com o processo de compilação. Parâmetros podem ser passados para o compilador através de comandos usados nessa seção. Normalmente, os programas possuem uma documentação sobre como proceder sua compilação e instalação. Essa seção deve seguir essas instruções de acordo com o que o desenvolvedor definiu.

Nos casos em que o programa oferece as facilidades das ferramentas autotools, é fornecido o *script* Makefile, responsável pela compilação do programa. Segundo (Bailey, 2000), "essa seção se resume a um comando make, ou algo mais complexo caso o programa requeira. Entretanto, a maioria dos programas atuais são preparados deste modo".

Para ilustrar o funcionamento da seção %build, foram anteriormente escritos os arquivos configure.in e makefile.in, A ferramenta autoconf é usada para gerar os *scripts* configure e Makefile. Isso torna o processo de compilação do programa-exemplo arl totalmente automático.

Assim, a seção `%build`, neste caso, se resume ao comando `make`, que fará o processo de compilação. A seguir, é apresentado um exemplo dessa seção:

```
# seção BUILD
%build
make
```

7.4.4 - Seção de instalação (`%install`).

Após concluído o processo de compilação do aplicativo, os arquivos executáveis gerados estão armazenados em um diretório usado para realizar esse processo. Esses arquivos não podem ser empacotados desta forma. É preciso criar uma hierarquia de diretórios idêntica à hierarquia que esses arquivos encontrarão ao serem instalados em um sistema GNU/Linux.

Entretanto, usar a estrutura de diretórios original do sistema hospedeiro, onde o pacote está sendo montado, não é uma idéia prática. Uma vez que não é recomendável construir pacotes como superusuário, então não é possível utilizar a estrutura de diretórios real do sistema. Além disso, proceder desta forma implicaria em questões de segurança, pois caso esse processo resultasse em erros o sistema hospedeiro ficaria danificado.

A solução é criar uma estrutura de diretórios virtual, porém idêntica à hierarquia de diretórios real que o programa encontrará ao ser instalado. O sistema RPM oferece duas variáveis de ambiente para realizar estas tarefas, cujos nomes são bastante semelhantes, mas com funções distintas.

A primeira variável é `$RPM_BUILD_DIR`, usada para referenciar o diretório onde o programa será compilado. Normalmente, o valor desta variável é `$(HOME)/rpmbuild/BUILD`, este é um diretório criado exatamente para receber os arquivos gerados durante o processo de compilação. Essa variável é criada automaticamente por RPM.

A segunda variável é `$RPM_BUILD_ROOT.` O valor desta variável é defi-

nido através do rótulo `Buildroot`, configurado na seção preâmbulo. O valor desta variável define um diretório raiz virtual. Nesta estrutura, o programa pode ser virtualmente "instalado" antes de ser empacotado.

De acordo com (Bailey, 2000), "nesta seção são incluídos os comandos necessários para instalar o programa. Caso o autor do programa tenha incluído um *script* de instalação, essa seção terá somente um comando `make install` ou `install`". Caso contrário, os processos serão feitos com comandos do *shell* relacionados com a manipulação de arquivos. A criação dos diretórios e a cópia dos aplicativos para seus devidos locais no caso do programa-exemplo `arl` serão feitas deste modo, como observado na listagem a seguir:

```
%install

rm -rf $RPM_BUILD_ROOT

mkdir -p $RPM_BUILD_ROOT$RPM_DOC_DIR/%{name}-%{version}

mkdir -p $RPM_BUILD_ROOT/usr/local/bin

cp $RPM_BUILD_DIR/%{name}-%{version}/arl $RPM_BUILD_ROOT/usr/lo-
cal/bin/arl

cp $RPM_BUILD_DIR/%{name}-%{version}/arl.html $RPM_BUILD_ROOT
$RPM_DOC_DIR/%{name}-%{version}/arl.html

cp $RPM_BUILD_DIR/%{name}-%{version}/rpm_logo.png $RPM_BUILD_ROOT
$RPM_DOC_DIR/%{name}-%{version}/rpm_logo.png
```

7.4.5 - Seção de "limpeza" (`%clean`)

Essa é uma seção usada apenas para apagar os arquivos incluídos na estrutura de diretórios virtual. Depois que os arquivos são empacotados, é recomendável limpar este diretório, essa seção será executada no fim do processo.

Durante testes de construção de pacotes, essa seção pode ser desativada temporariamente. Um exemplo pode ser visto a seguir:

```
%clean
rm -rf $RPM_BUILD_ROOT
```

7.4.6 - Seção de arquivos (`%files`)

Após o processo de instalação, os arquivos executáveis, bibliotecas e documentação estão prontos para serem empacotados. Entretanto, nem todos os arquivos gerados ou usados durante a compilação farão parte do mesmo pacote. Cada arquivo gerado durante a compilação irá para um tipo de pacote específico de acordo com sua finalidade ou mesmo não serão empacotados.

Assim, arquivos binários, de configuração e documentação irão para o pacote binário, enquanto arquivos de código-fonte, *patches*, *scripts* do tipo *System V Init* irão para o pacote fonte. Arquivos de código-fonte de bibliotecas e cabeçalhos usados pelas linguagens `C` ou `C++` podem ser incluídos em um pacote de desenvolvimento. Existe ainda o pacote de depuração (*debug*) criado automaticamente por RPM. Além disso, é possível criar sub-pacotes binários.

Segundo (Guru Labs, 2005), “essa seção lista os arquivos e diretórios que devem ser incluídos em cada pacote. É listado um ou mais arquivos por linha e caracteres “curingas” podem ser usados”. As rotas definidas nessa seção são relativas ao diretório definido na variável `$RPM_BUILD_ROOT`, sendo que cada valor que representa um caminho completo até um arquivo ou diretório será ajustado automaticamente por RPM. Assim, cada rota deve começar com uma barra, indicando o diretório raiz real.

Diversas macros podem ser usadas nessa seção. A macro `%defattr` é usada para definir propriedades de dono e grupo e configurar as permissões dos arquivos ou diretórios listados após está macro. Para configurar individualmente

um arquivo ou diretório, deve ser usada a macro `%attr`.

De acordo com (Ewing; *et al*, 2002), as macros listadas a seguir são usadas para definir o tipo de um arquivo. Assim, durante uma consulta, é possível filtrar os arquivos por seu tipo. Essas macros são assim definidas:

- `%config` - Define que o arquivo é de configuração. Usada com o parâmetro (`noreplace`), indica que este arquivo deve ser mantido inalterado no caso de uma atualização;
- `%doc` - Indica que o arquivo é de documentação;
- `%license` - Indica que o arquivo é uma licença;
- `%readme` - Indica que o arquivo possui instruções que devem ser lidas;
- `%ghost` - Indica que o conteúdo deste arquivo não será incluído no *payload* do pacote;
- `%dir` - Indica um diretório.

Um exemplo desta seção pode ser visto a seguir:

```
%files
%defattr (0770,root,root)
/usr/local/arl-2004.2/bin/arl
%doc /usr/local/arl-2004.2/doc/arl.html
%doc /usr/local/arl-2004.2/doc/rpm_logo.png
```

7.4.7 - Seção histórico de mudanças (`%changelog`)

Segundo (Guru Labs, 2005), esta seção armazena um histórico com as alterações feitas pelos desenvolvedores do pacote. Deste modo, é possível ter um acompanhamento da evolução do desenvolvimento do aplicativo. Cada entrada nessa seção deve seguir uma sintaxe pré-definida, como a seguir:

```
#Comentário sobre o changelog
* Data da alteração empacotador <e-mail de contato> versão-release
- Descrição da alteração
- Descrição de outras alterações
```

Cada nova entrada, deve ser incluída no começo do arquivo. Um exemplo desta seção pode ser visto a seguir:

```
%changelog
# Lançamento
* Sat Aug  5 2006 14:18:08 unasi <unasi.arl@nospam.com>
- Retirado o bug da linha extra (#1)
- Refeito o arquivo spec para apresentação
```

7.4.8 - Seções preparatórias ou opcionais.

Além destas seções consideradas básicas, existem outras seções consideradas avançadas e usadas para ajustar o ambiente antes ou após uma instalação ou remoção. Segundo (Guru Labs, 2005), algumas vezes, é necessário executar comandos em um sistema antes ou após um programa ser instalado. Essas seções são listadas após a seção "arquivos" e são simplesmente *scripts do shell* usando comandos de terminal para realizar tarefas diversas, por exemplo, criar uma conta de usuário para um programa ou remover algum diretório que não é mais necessário após a desinstalação. São seções opcionais:

- `%pre` - executada antes da instalação;
- `%post` - executada após a instalação;
- `%preun` - executada antes da desinstalação;
- `%posun` - executada após a desinstalação.

Outras duas seções opcionais são utilizadas para serem executadas em modo condicional. Isso significa que, para serem executadas, é necessário que uma condição qualquer seja verdadeira ou falsa. Por exemplo, um sistema que interage com o aplicativo de envio de correio eletrônico pode ter um comportamento diferente se o sistema instalado for `Sendmail` ou `Postfix`.

Assim, estas seções "gatilho" serão disparadas em função do ambiente em que o pacote for instalado. Essas seções são:

- `%triggerin` - executada quando o pacote é instalado ou atualizado;
- `%triggerrun` - executada quando o pacote é removido.

7.5 - O arquivo-exemplo arl.spec.

Uma listagem completa do arquivo arl.spec é apresentada a seguir:

```
Summary: ARL - Administração em Redes Linux - Pós-Graduação/ Lato Sensu.
Name:arl
Version:2004.2
Release:1
License:GPL
Group:"Pós-Graduação/Lato Sensu"
URL:http://www.ginux.comp.ufla.br/arl
Vendor:UFLA/FAEPE
Packager:Keynes Augusto <unasi@nospam.com>
Source0:%{name}-%{version}.tar.gz
Patch0: arl.fix1.patch
Prefix:/usr/local/bin
Prefix:/usr/share/doc
Buildroot: %{_tmppath}/%{name}-%{version}-root

%description
Este é o pacote RPM para o famoso ARL. ARL é um comando usado para exibir
a data e a hora local de um sistema GNU/Linux.

# Seção PREP
%prep
%setup
cd %{_topdir}
%patch -P 0
cd $RPM_BUILD_DIR/%{name}-%{version}
./configure

# Seção BUILD
```

```
%build
make

# Secão INSTALL
%install

rm -rf $RPM_BUILD_ROOT
mkdir -p $RPM_BUILD_ROOT$RPM_DOC_DIR/%{name}-%{version}
mkdir -p $RPM_BUILD_ROOT/usr/local/bin
cp $RPM_BUILD_DIR/%{name}-%{version}/arl $RPM_BUILD_ROOT/usr/local/bin/arl

cp $RPM_BUILD_DIR/%{name}-%{version}/arl.html
$RPM_BUILD_ROOT$RPM_DOC_DIR/%{name}-%{version}/arl.html

cp $RPM_BUILD_DIR/%{name}-%{version}/rpm_logo.png
$RPM_BUILD_ROOT$RPM_DOC_DIR/%{name}-%{version}/rpm_logo.png

#Seção CLEAN
%clean
rm -rf $RPM_BUILD_ROOT

# Seção FILES
%files
%defattr (-,root,root)
/usr/local/bin/arl
%doc /usr/share/doc/%{name}-%{version}/arl.html
%doc /usr/share/doc/%{name}-%{version}/rpm_logo.png

# Seção CHANGELOG
%changelog
# Comentário sobre o changelog
* Sat Aug  5 2006 14:18:08 unasi <unasi.arl@nospam.com>
- Retirado o bug da linha extra (#000001)
- Refeito o arquivo spec para apresentação
```

8 - O ambiente de produção de pacotes RPM

8.1 – Geração de um par de chaves públicas

Anteriormente, foram demonstrados os processos de importação de uma chave pública para o chaveiro de RPM. Deste modo, pacotes obtidos de fornecedores, cuja chave pública estivesse no chaveiro de RPM, poderiam ser checados quanto a autenticidade e integridade através das opções de segurança.

Nesta seção, será demonstrado como criar um par de chaves, uma pública e outra privada, que serão usadas para assinar e checar as assinaturas dos pacotes produzidos neste trabalho. A função destas chaves é assinar digitalmente os pacotes garantindo a integridade de seu conteúdo e a autenticidade do pacote.

A assinatura de pacotes não é obrigatória, entretanto, por motivos óbvios de segurança é recomendado usar somente pacotes que possam ser checados através de suas assinaturas digitais. O risco de instalar um programa com código malicioso é considerável. Desta maneira, faz parte do conjunto de boas políticas de um administrador trabalhar preferencialmente com pacotes assinados e de fornecedores que sejam comprovadamente seguros.

Ao desenvolver pacotes, o correto é fornecer uma chave pública para que os usuários de seus pacotes possam proceder a checagem da sua autenticidade e da sua integridade. De acordo com (Guru Labs, 2005), a questão das assinaturas digitais de pacotes pode ser assim descrita:

> RPM fornece a capacidade de assinar digitalmente os pacotes gerados usando a tecnologia de criptografia de chave pública de GPG. Assinaturas digitais são importantes para que os usuários dos seus pacotes possam checar a integridade e autenticidade dos mesmos. Assim, se certificam que não esta instalando um pacote que possui código malicioso criado por um usuário que se faz passar por um produtor de pacotes.

Detalhes do funcionamento do GnuPG ou sobre criptografia de chave pública estão fora do escopo deste trabalho. Mais informações sobre GnuPG podem ser obtidas na documentação de GnuPG. Informações sobre a tecnologia de criptografia de chave pública podem ser obtidas em (Tanenbaum, 2003).

Uma vez que não é recomendável gerar pacotes autenticado como o superusuário, é preciso realizar uma série de procedimentos para ajustar o ambiente de um usuário comum que fará o papel de empacotador. Esses ajustes são para configurar GnuPG e para criar o par de chaves pública e privada.

A primeira vez que GnuPG é executado por um usuário comum, automaticamente é criada uma estrutura de diretórios para abrigar as chaves e os chaveiros deste usuário. Basta invocar o programa **gpg**, como no exemplo seguinte:

```
$ gpg
gpg: directory `/home/empacotador/.gnupg' created
gpg: criado novo ficheiro de configuração `/home/empacotador/.gnupg/gpg.-
conf'
gpg: AVISO: opções em `/home/empacotador/.gnupg/gpg.conf' ainda não estão
activas nesta execução
gpg: porta-chaves `/home/empacotador/.gnupg/secring.gpg' criado
gpg: porta-chaves `/home/empacotador/.gnupg/pubring.gpg' criado

gpg: Vá em frente e digite sua mensagem ...

(CTRL + C)

gpg: Interrupt caught ... exiting
```

Após invocar o programa **gpg**, é criado o diretório oculto `.gnupg`, o arquivo personalizado de configuração `gpg.conf` para o usuário e os arquivos chaveiros `pubring.gpg` e `secring.gpg`, respectivamente, para armazenar as chaves públicas e privadas deste usuário. Após o aviso do programa **gpg** que diz: "`Vá em frente e digite sua mensagem ...`" o usuário deve interromper a execução de **gpg** teclando simultaneamente CTRL e C (*control* + C). Criada a estrutura de diretórios para GnuPG, o próximo passo é criar as chaves, de acordo com

as instruções a seguir:

a) Invocar o programa `gpg` com a opção `--gen-key;`

b) Escolher o tipo de criptografia desejada. Atualmente, é recomendada a opção "`DSA e Elgamal`" por questões de qualidade do algoritmo;

c) Escolher o tamanho da chave. Atualmente, 2048 bits é uma boa escolha, sendo inclusive a sugestão padrão de GnuPG;

d) Escolher o tempo de validade da chave. Chaves de autenticação de pacotes não devem expirar. Então, escolher a opção "`a chave não expira`";

e) Confirmar se os dados estiverem corretos;

f) Escolher uma identificação para a chave. Uma identificação é um campo formado pelo nome completo do empacotador, um comentário e um endereço de correio eletrônico para contato.

Este campo terá o seguinte formato:

```
Nome Completo (Comentário) <e-mail@nospam.com>
```

Digitar as opções e confirmar no final com (O)K.

g) Este passo envolve a escolha de uma frase secreta que funciona como uma senha a ser pedida sempre que algum pacote precise ser assinado. A frase secreta evita que terceiros que indevidamente tenham obtido a chave privada possam usá-la para assinar algo. O processo de criação da chave pode ser visto a seguir:

```
$ gpg --gen-key
Por favor selecione o tipo de chave desejado:
   (1) DSA and Elgamal (default)
   (2) DSA (apenas assinatura)
   (5) RSA (apenas assinatura)
Sua opção? 1
DSA keypair will have 1024 bits.
ELG-E keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
O tamanho de chave pedido é 2048 bits
Por favor especifique por quanto tempo a chave deve ser válida.
         0 = chave não expira
      <n>  = chave expira em n dias
```

```
      <n>w = chave expira em n semanas
      <n>m = chave expira em n meses
      <n>y = chave expira em n anos
A chave é valida por? (0) 0
Key does not expire at all
Is this correct? (y/N)
You need a user ID to identify your key; the software constructs the user
ID
from the Real Name, Comment and Email Address in this form:
   "Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"
Nome completo: Keynes Augusto
Endereço de correio eletrônico: unasi@nospam.com
Comentário: Unasi
Você selecionou este identificador de usuário:
    "Empacotador (Unasi) <unasi@nospam.com>"

Muda (N)ome, (C)omentário, (E)ndereço ou (O)k/(S)air? O
Você precisa de uma frase secreta para proteger sua chave.

Precisamos gerar muitos bytes aleatórios. É uma boa idéia realizar outra
atividade (digitar no teclado, mover o mouse, usar os discos) durante a
geração dos números primos; isso dá ao gerador de números aleatórios
uma chance melhor de conseguir entropia suficiente.
gpg: /home/empacotador/.gnupg/trustdb.gpg: banco de dados de confiabilida-
de criado
gpg: key D411BE4A marked as ultimately trusted
chaves pública e privada criadas e assinadas.

gpg: a verificar a base de dados de confiança
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
pub   1024D/D411BE4A 2006-08-14
      Key fingerprint = 722F D688 9C90 A874 A3FC  DFE2 0485 841D D411 BE4A
uid                  Keynes Augusto (Unasi) <unasi@nospam.com>
sub   2048g/A4653366 2006-08-14
```

h) Verificar a geração da chave com a opção `--list-keys`:

```
$ gpg --list-keys

/home/empacotador/.gnupg/pubring.gpg
---------------------------------
pub   1024D/D411BE4A 2006-08-14
uid                  Empacotador (Unasi) <unasi@nospam.com>
sub   2048g/A4653366 2006-08-14
```

i) O passo seguinte é exportar essa chave para um arquivo (`gpg-public-key.asc`), em um formato que possa ser importada pelo chaveiro de RPM. Somente após esse processo, é possível verificar pacotes assinados com esta chave.

```
$ gpg --export --armor > /home/empacotador/gpg-public-key.asc
```

j) Importação da chave para o chaveiro de RPM.

A importação de uma chave pública para o chaveiro de RPM é uma operação exclusiva do superusuário por questões de segurança.

```
# rpm --import /home/empacotador/gpg-public-key.asc
```

8.2 - Configuração do ambiente de produção de RPM.

Inicialmente, o ambiente de RPM é preparado para que o superusuário possa construir pacotes. No sistema Fedora Core, a estrutura de diretórios armazenada sob `/usr/src/redhat` é usada para a construção de pacotes pelo superusuário. Todavia, não é recomendável a construção de pacotes usando o superusuário por diversas questões de segurança envolvendo o sistema hospedeiro.

Em função dessas características, é preciso configurar o ambiente para que um ou mais usuários comuns possam realizar a tarefa de empacotar programas. O pacote `fedora-rpmdevtools` oferece o *script* `fedora-buildrpmtree` que realiza automaticamente a configuração do ambiente para o usuário. Basicamente, as configurações feitas pelo *script* são:

- Criação do diretório `rmpbuild` na pasta particular do usuário e dos subdiretórios `BUILD, SOURCES, SPECS, RPMS e SRMPS.` Utilizados para abrigar os arquivos usados durante a construção e também os pacotes gerados nesse processo;

- Criação do arquivo .rpmmacros no diretório particular do usuário, com os ajustes na macro `%_topdir`, apontando para a estrutura de diretórios em `%(echo HOME)/rpmbuild/.` As configurações neste arquivo vão se sobrepor às configurações globais. As configurações referentes ao uso de chave pública devem ser feitas manualmente, para que seja possível assinar pacotes usando GnuPG.

Caso queira, o usuário poderá obter os mesmos ajustes através de comandos manuais para a criação dos diretórios e arquivos, como no exemplo a seguir:

a) Criação da estrutura de diretórios para construção de pacotes:

```
$  mkdir -p rpmbuild/{SOURCES,SPECS,BUILD,SRPMS,RPMS}

$  mkdir rpmbuild/RPMS/{i386,i686}
```

b) Criação do arquivo `.rpmmacros` para o usuário com o seguinte conteúdo:

```
%_topdir      %(echo $HOME)/rpmbuild
% debug_package %{nil}
%_smp_mflags  -j3
%__arch_install_post   /usr/lib/rpm/check-rpaths /usr/lib/rpm/ \
check-buildroot
%_signature gpg
%_gpg_path /home/empacotador/.gnupg
%_gpg_name Empacotador (Unasi) <unasi@nospam.com>
%_gpgbin /usr/bin/gpg
```

Segundo (Guru Labs, 2005), incluindo estas opções no arquivo `.rpmmacros`, o usuário poderá assinar os pacotes gerados por RPM.

9 – Construção de pacotes com o utilitário rpmbuild.

Até a versão 3.x, o comando `rpm` realizava as tarefas relacionadas com a administração de pacotes e com a geração dos pacotes. A partir da versão 4.x, essas tarefas foram adequadamente distribuídas entre alguns utilitários, de acordo com a função a ser realizada. Desta maneira, em sistemas modernos (versões maiores que 4.x), o utilitário responsável pela geração dos pacotes é denominado `rpmbuild`. Segundo (Ewing; *et al*, 2002), rpmbuild é usado tanto para a construção de pacotes binários quanto pacotes fonte.

Para o utilitário rpmbuild, cada seção de um arquivo *.spec* é considerada um estágio da construção de um pacote. Assim, é possível manipular a construção de um pacote executando as seções em uma certa ordem, conforme elas vão sendo escritas e resolvidas. A sintaxe geral de `rpmbuild` é :

```
rpmbuild -bSTAGE|-tSTAGE [rpmbuild-options] SPEC.
```

Os estágios definidos com a opção **-b** (*build*) são usados quando há um arquivo *.spec* externo. Os estágios usados com a opção **-t** (*tarball*) são usados quando o desenvolvedor disponibilizou junto ao arquivo do tipo **.tar.gz** um arquivo do tipo *.spec,* que também auxilia a construção de pacotes.

Normalmente, as opções de construção (*build*) são mais usadas. O mais comum é construir pacotes binários a partir de pacotes fonte do tipo SRPM.

As opções de construção são definidas no Quadro 5, de acordo com as informações de (Ewing; *et al*, 2002).

Quadro 5 - Opções de construção de pacotes de rpmbuild.

Opção	Opção	Estágios Executados	Pacotes Gerados
-ba	-ta	%prep, %build e %install	Binário e Fonte.
-bb	-tb	%prep, %build e %install	Somente Binário.
-bp	-tp	%prep	Nenhum.
-bc	-tc	%prep, %build	Nenhum.
-bi	-ti	%prep, %build e %install	Nenhum.
-bl	-tl	Faz um *check-list* em %files	Nenhum.
-bs	-ts	Nenhum	Somente Fonte.

Essas opções podem ser usadas durante a depuração de um arquivo do tipo *.spec.* Deste modo, a sintaxe pode ser checada passo a passo, em cada um dos estágios de construção do pacote. As mesmas opções são válidas para arquivos do tipo *tarball*[37] que possuem um arquivo *.spec* incluído e que são construídos com as opções **-t**, idênticas às opções **-b**.

Além das opções de estágios de construção, podem ser utilizados parâmetros opcionais chamados de `rpmbuild-options`. A função destas opções é passar parâmetros para `rpmbuild` modificando valores de opções que normalmente são definidas no próprio arquivo *.spec*. A mudança destes parâmetros modifica o comportamento da construção do pacote. De acordo com (Ewing; *et al*, 2002), a sintaxe destas opções é :

--buildroot Directory

O valor passado em Directory, sobrescreve o valor do rótulo Buildroot definido no arquivo *.spec*.

37 Um arquivo *tarball* é gerado pelo utilitário **tar** com opção de compactação por **GZip**. Possuem extensão .tar.gz.

--clean

Remove a árvore de diretórios onde os arquivos foram produzidos, após os pacotes terem sido gerados.

--norebuild

Não executa novamente o estágio *build*. Usado para testar arquivos *.spec*.

--rmsource

Remove os arquivos fonte do diretório SOURCES após a construção do pacote.

--rmspec

Remove o arquivo *.spec* do diretório SPECS após a construção do pacote.

--short-circuit

Salta a execução de determinados estágios. Válido somente com as opções **-bc** e **-bi**, permite continuar a construção a partir do próximo estágio onde havia parado anteriormente.

--sign

Inclui em tempo de execução uma assinatura GPG no pacote. Essa assinatura também pode ser incluída posteriormente usando o utilitário `rpmsign`.

--target `Platform`

Permite definir uma plataforma para a construção do pacote. A plataforma deve ser compatível com o processador do hospedeiro. Não é possível construir pacotes para uma plataforma incompatível com seu processador.

9.1 – Utilização de pacotes fonte SRPM

Segundo (Foster-Johnson, 2005), "a maior parte do trabalho com `rpmbuild` será para criar pacotes binários a partir de um arquivo *tarball* e um *.spec*. Todavia, também é possível obter pacotes fonte SRPM preparados pelo distribuidor do sistema e usá-los para gerar um pacote binário".

Um pacote fonte SRPM é como qualquer outro pacote binário RPM do

ponto de vista estrutural. É possível utilizar as opções de consulta para extrair informações diretamente desses pacotes e realizar a sua instalação. Porém, a instalação de um pacote fonte do tipo SRPM é um tanto diferente da instalação de um pacote binário RPM.

Normalmente, um pacote binário implementa arquivos, em uma série de diretórios, que juntos vão formar um aplicativo. No caso dos pacotes fonte SRPM, os arquivos implementados são unicamente um arquivo do tipo *tarball*, alguns arquivos com *patches* e *scripts* e um arquivo do tipo *.spec*. Ou seja, o arquivo fonte SRPM instala no sistema os requisitos para a construção de pacotes.

Desta maneira, quando instalados, pacotes fonte SRPM são alocados na estrutura de diretórios criada para a construção de pacotes. Os arquivos *tarball,* os *scripts* e os *patches* são alocados no diretório SOURCES, enquanto o arquivo *.spec* é salvo no diretório SPECS. Informações sobre pacotes fonte não são salvas na base de dados RPM. As consultas são feitas diretamente ao arquivo pacote. Normalmente, após serem usados para a construção de pacotes, esses arquivos são apagados do sistema.

O sistema RPM oferece dois modos de trabalhar com pacotes fonte. Pode ser usado o modo `--rebuild` ou `--recompile` de acordo com a finalidade desejada. Segundo (Bailey, 2000), os dois modos de operação são quase praticamente idênticos, a única diferença é descrita do seguinte modo:

> O modo `--rebuild` faz as seguintes tarefas: instala o pacote fonte, descompacta os fontes, compila, instala o programa no sistema virtual e limpa o diretório onde a aplicação foi construída. Além disso, a construção de um pacote binário também é realizada. Enquanto a opção `--recompile` também faz todos esses passos mas não gera nenhum pacote essa é a única diferença entre os dois modos.

Então, na verdade, esses dois modos são vistos como atalhos para simplificar as operações de compilação e construção de pacotes.

9.2 – Construção dos pacotes para o programa-exemplo

Baseado na teoria sobre construção de pacotes RPM, vista neste trabalho, a demonstração prática da construção dos pacotes do programa-exemplo ARL possibilita testar os conhecimentos adquiridos. Antes de iniciar a construção de pacotes RPM, é preciso certificar-se da existência dos requisitos e das configurações necessárias.

De acordo com (Guru Labs, 2005), os seguintes pacotes devem estar instalados e são básicos para o desenvolvimento de programas: `gcc`, `gcc-c++`, `make, bison e binutils`. Além destes, as bibliotecas de desenvolvimento usadas também devem estar presentes. Por estar fora do escopo deste trabalho indicar passo-a-passo a construção de um ambiente de desenvolvimento, o modo mais simples de obter este ambiente é instalar o sistema GNU/Linux com o perfil de desenvolvimento. As principais distribuições GNU/Linux possuem este perfil para instalação.

Além disto, a partir da versão `4.x`, o sistema RPM passou a ser distribuído em uma série de pacotes de acordo com as finalidades dos utilitários. O utilitário encarregado de proceder a construção dos pacotes é `rpmbuild`, que é instalado pelo pacote `rpm-build`. Portanto, este pacote deve ser instalado no sistema.

Outro pacote que deve ser instalado, no caso da distribuição Fedora Core, é o pacote `fedora-rpmdevtools` que implementa vários utilitários usados na construção de pacotes.

Conforme afirmado anteriormente, não é recomendável construir pacotes usando o superusuário. Deste modo, o ambiente de construção de pacotes deve ser configurado para um usuário comum, seguindo as instruções do capítulo 8 deste trabalho. Configurado o ambiente de produção, basta alocar o arquivo *tarball* e os *patches* no diretório SOURCES, além do arquivo *.spec* no diretório SPECS para iniciar os testes de construção dos pacotes.

Seguindo as instruções do capítulo 9, é possível proceder a compilação dos arquivos e a criação dos pacotes fonte e binário com a seguinte sintaxe:

```
$ rpmbuild -ba --target=i686 --sign arl.spec
```

Depois de construídos, o pacote binário é alocado automaticamente no diretório em `~/rpmbuild/RPMS/i686` e o pacote fonte, em `~/rpmbuild/SRPMS`. Ambos estão prontos para serem usados com os vários modos de RPM. A seguir, é apresentada uma série de exemplos de comandos RPM aplicados aos pacotes fonte e binário criados, demonstrando o correto funcionamento da construção de pacotes:

a) Exibição dos nomes dos arquivos que são implementados pelo pacote:

```
$ rpm -qlp arl-2004.2-1.i686.rpm
/usr/local/bin/arl
/usr/share/doc/arl-2004.2/arl.html
/usr/share/doc/arl-2004.2/rpm_logo.png
```

b) Exibição apenas dos arquivos de documentação implementados pelo pacote:

```
$ rpm -qdp arl-2004.2-1.i686.rpm
/usr/share/doc/arl-2004.2/arl.html
/usr/share/doc/arl-2004.2/rpm_logo.png
```

c) Exibição das informações do preâmbulo do pacote binário `arl-2004.2-1.i686.rpm`:

```
$ rpm -qip arl-2004.2-1.i686.rpm
Name        : arl                       Relocations: /usr/local/bin /usr/share/doc
Version     : 2004.2                    Vendor: UFLA/FAEPE
Release     : 1                         Build Date: Qua 16 Ago 2006 14:02:17 BRT
Install Date: (not installed)           Build Host: kye2400.kyetoy.net
Group       : "Pós-Graduação/Lato Sensu"   Source RPM: arl-2004.2-1.src.rpm
Size        : 18817                     License: GPL
Signature   : DSA/SHA1, Qua 16 Ago 2006 14:02:18 BRT, Key ID 4c4edad224f0a81a
Packager    : Keynes Augusto <unasi@nospam.com>
URL         : http://www.ginux.comp.ufla.br/arl
Summary     : ARL - Administração em Redes Linux - Pós-Graduação/Lato Sensu
Description :
Este é o pacote RPM para o famoso ARL. ARL é um comando usado para exibir a data e a
hora local de um sistema GNU/Linux.
```

d) Checagem da integridade e autenticidade do pacote:

```
$ rpm -K arl-2004.2-1.i686.rpm
arl-2004.2-1.i686.rpm: (sha1) dsa sha1 md5 gpg OK
```

e) Listagem do *changelog* do pacote:

```
$ rpm -qp --changelog arl-2004.2-1.i686.rpm
* Sáb Ago 05 2006 14:18:08 unasi <unasi.arl@nospam.com>
- Retirado o bug da linha extra (#1)
- Refeito o arquivo spec para apresentaçãoarl-2004.2-1.i686.rpm:
(sha1) dsa sha1 md5 gpg OK
```

f) Exibição das informações internas do pacote:

```
$ rpm -qp --dump arl-2004.2-1.i686.rpm

/usr/local/bin/arl 5343 1155747737 48d50807119efb70bb9f008774b8a6e3 0100755 root
root 0 0 0 X
/usr/share/doc/arl-2004.2/arl.html 886 1155747737 8156e80a74fa960c329e0f25b784bde4
0100644 root root 0 1 0 X
/usr/share/doc/arl-2004.2/rpm_logo.png 12588 1155747737
9bb1c3ad707160d6eb36619009268be3 0100644 root root 0 1 0 X
```

g) Exibição dos arquivos instalados pelo pacotes associados ao nome do pacote:

```
$ rpm -qp --filesbypkg arl-2004.2-1.i686.rpm
arl                       /usr/local/bin/arl
arl                       /usr/share/doc/arl-2004.2/arl.html
arl                       /usr/share/doc/arl-2004.2/rpm_logo.png
```

h) Exibição dos requerimentos (dependências) do pacote:

```
$ rpm -qp --requires arl-2004.2-1.i686.rpm
libc.so.6
libc.so.6(GLIBC_2.0)
libgcc_s.so.1
libm.so.6
libstdc++.so.6
libstdc++.so.6(CXXABI_1.3)
rpmlib(CompressedFileNames) <= 3.0.4-1
rpmlib(PayloadFilesHavePrefix) <= 4.0-1
```

i) Instalação do pacote binário:

```
# rpm -Uvh arl-2004.2-1.i686.rpm
A preparar...
########################################### [100%]
   1:arl
########################################### [100%]
```

j) Verificação do pacote instalado:

```
# rpm -Vv arl
.........   /usr/local/bin/arl
......... d /usr/share/doc/arl-2004.2/arl.html
......... d /usr/share/doc/arl-2004.2/rpm_logo.png
```

k) Exibição dos provimentos do pacote:

```
# rpm -q --whatprovides arl
arl-2004.2-1
```

l) Lista a data e a hora de instalação do pacote:

```
# rpm -q --last arl
arl-2004.2-1                    Qua 16 Ago 2006 15:01:25 BRT
```

m) Exibe o estado dos arquivos instalados pelo pacote:

```
# rpm -qs arl
normal        /usr/local/bin/arl
normal        /usr/share/doc/arl-2004.2/arl.html
normal        /usr/share/doc/arl-2004.2/rpm_logo.png
```

n) Exibição dos arquivos do pacote fonte:

```
# rpm -qpl arl-2004.2-1.src.rpm
arl-2004.2.tar.gz
arl.fix1.patch
arl.spec
```

10 - Conclusão

A instalação de aplicativos em sistemas GNU/Linux é uma tarefa complexa que requer amplos conhecimentos sobre compilação de programas e sobre a estrutura de diretórios do sistema. No passado, diversos sistemas do tipo Unix possuíam ferramentas para facilitar a gerência de programas instalados. Todavia, essas tentativas de desenvolvimento de sistemas de gerenciamento de pacotes não foram bem sucedidas por diversos motivos, tais como, não preservavam o código-fonte e não possuíam suporte para múltiplas arquiteturas. Desta maneira, esses aplicativos deixam a desejar e não se tornaram um padrão *de facto* como RPM acabou sendo.

Baseada nestas dificuldades, a empresa Red Hat planejou e financiou o desenvolvimento de um sistema de empacotamento que tivesse como filosofia a facilidade de uso tanto para usuários quanto para desenvolvedores de pacotes. O sistema RPM foi projetado por especialistas que reuniram o conhecimento acumulado com os erros de desenvolvimento dos sistemas anteriores e objetivaram solucionar a maior parte dos problemas encontrados nessas ferramentas.

O sistema RPM fornece um conjunto de métodos para manipular pacotes, mantendo as informações em uma base de dados que pode ser consultada. Diversas características vistas ao longo deste trabalho demonstram que RPM é um sistema bastante funcional e flexível do ponto de vista dos administradores de sistemas GNU/Linux.

A gama de opções para consulta permite manter um estrito controle sobre os sistemas instalados. Diversas opções de uso permitem manipular pacotes procedendo a instalação, atualização e remoção com um simples comando.

Do ponto de vista do gerenciamento de programas distribuídos neste formato, RPM se tornou o padrão *de facto* para a indústria de distribuição de aplicativos. Além das propriedades do próprio sistema RPM, o gerenciamento de pro-

gramas ainda conta com as ferramentas de atualização como o apt-get e o Yum. A maioria das tarefas pode ser feita através de interfaces gráficas que facilitam a gestão dos programas instalados.

As facilidades para o gerenciamento implantadas por RPM permitem um ganho de produtividade, se comparadas aos métodos tradicionais de instalação e manutenção de sistemas. É possível manter um sistema sempre atualizado com as últimas versões de segurança ou de aprimoramento com um grau de esforço muito pequeno.

Na administração de sistemas GNU/Linux moderna, o uso de pacotes é altamente recomendado. Os principais motivos para usar o sistema RPM de pacotes são agilidade e segurança. O uso de pacotes evita a complexidade do ambiente de desenvolvimento e reduz as chances de erros a um nível praticamente nulo. É muito difícil imaginar alguém administrando um ambiente com várias máquinas utilizando o método manual de instalação e manutenção. O sistema RPM oferece a desenvoltura, agilidade e segurança que os administradores precisam para realizar suas tarefas de administração de programas.

Do ponto de vista do desenvolvimento de pacotes, RPM fornece as ferramentas para que o administrador tenha a liberdade de trabalhar com pacotes fonte e reconstruir ou recompilar, de modo personalizado, aplicativos inteiros apenas com uma linha de comando, sem qualquer complicação técnica ou teórica.

A alta padronização das tarefas de produção de pacotes permite criar distribuições inteiras com qualidade, além de permitir a atualização dos pacotes com o mínimo de esforço dos empacotadores.

Grande parte do sucesso do sistema RPM se deve ao fato de estar acoplado às ferramentas de desenvolvimento denominadas `autotools`. Esse conjunto de aplicativos facilita a preparação, compilação e instalação de programas no sistema GNU/Linux. O sistema RPM utiliza intensamente essas ferramentas para construir seus pacotes binários.

Então, seja na visão da gerência de sistemas ou na visão de desenvolvimento de pacotes, RPM é um sistema maduro cujo desenvolvimento aponta para uma estabilidade ainda maior. Se pode observar, pelos vários autores consultados, que foram elogiosos para com a ferramenta e ressaltaram seus benefícios, importância e as facilidades oferecidas pelo sistema na gerência de programas.

A aprendizagem do uso de RPM é conseguida com um treinamento rápido. A forma como o sistema é idealizado permite que, aprendendo alguns poucos comandos, seja possível trabalhar eficientemente com a ferramenta.

Usuários avançados e desenvolvedores terão suporte das várias opções de uso que permitem integrar RPM a aplicativos e personalizar as consultas, de modo que o ambiente seja checado de acordo com as necessidades pessoais de cada administrador.

Desta maneira, a indicação do uso RPM para administradores de sistemas GNU/Linux é fato consumado. Os benefícios no uso do sistema são amplos, a documentação do sistema é consistente e seu desenvolvimento é permanente.

O domínio de uma ferramenta como RPM representa mais segurança, estabilidade e praticidade para administradores que estão envolvidos na gerência de ambientes com muitas máquinas ou que requerem um controle restrito sobre os programas instalados.

REFERÊNCIAS BIBLIOGRÁFICAS

BAILEY, Edward C. - **Maximum RPM**, [on-line] , 2000. Copyright Red Hat, Inc. Disponível em PDF na Internet em: <http://www.rpm.org/max-rpm/>

CAMARGO, Herlon A. - **Automação de Tarefas** - Lavras, MG – Editora. UFLA/FAEPE – 2005 - Capítulo 5 – páginas. 125-138

DÍGITRO, Tecnologia. - **Glossário tecnológico**, [on-line], 2006. Coordenação Eng. Djan de Almeida do Rosário, desenvolvida por Adm. Cláudio Brancher Kerber, apresenta termos tecnológicos na área de telecomunicações. Disponível na Internet em : <http://www.portaldigitro.com.br/novo/glossario_digitro.php>

EWING, Marc; JOHNSON, Jeff; TROAN, Erik – **RPM – Documentação.** [toda documentação disponível em /usr/share/doc/rpm-4.4.2 incluindo os manuais man] – Copyright Red Hat Linux - 2002

FOSTER-JOHNSON, Eric - **RPM Guide**, [on-line] 2005.Copyright 2003-2005. Red Hat, Inc. Disponível na Internet em: <http://fedora.redhat.com/docs/drafts/rpm-guide-en/index.html>

GURU LABS - **Creating Quality RPMs**, [on-line] 2005. Copyright Guru labs, sob a licença Creative Commons. Disponível na Internet em: <http://www.guru-labs.com/GURULABS-RPM-LAB/GURULABS-RPM-GUIDE-v1.0.PDF>

INTEL, Corporation. - **Informações sobre processadores Intel**, [on-line] 2006. Copyright Intel Corporation, Disponível na Internet em: <http://www.intel.-com/portugues/products/processor_number/definitions.htm>

LSB, Linux Standard Base - **LSB - versão 1.2 da especificação**, [on-line] Disponível na Internet em: <http://www.linuxbase.org/spec/refspecs/LSB_1.2.0/gLSB/book1.html>

MORIMOTO, Carlos E. - **Linux, Entendendo o Sistema: Guia Prático**, Porto Alegre, RS, Editora. Sul Editores, 2005 , 1a Edição.

NEMETH, Evi - **Manual completo do Linux** - São Paulo, SP, Editora Pearson Makron Books, 2004 1a Edição.

RIBEIRO, Uirá – **Certificação Linux** – Rio de Janeiro, RJ, Editora. Axcel Books, 2004, 1a Edição.

SILVA, Gleydson M. da - **Guia Foca Linux - Edição 5.45 da Versão Intermediário**, [on-line] 2006. Copyleft 1999-2006 Gleydson M. Da Silva. Disponível na Internet em: <http://focalinux.cipsga.org.br/guia/intermediario/index .htm>

TANENBAUM, Andrew S. - **Redes de Computadores**, Tradução Vandenberg D. de Souza. - Rio de Janeiro, RJ , Editora. Campus-Elsevier, 2003. 4a Edição.

TANENBAUM, Andrew S. - **Sistemas Operacionais Modernos**, Tradução Prof. Dr. Ronaldo A. L. Gonçalves e Prof. Dr. Luís A Consularo – São Paulo, SP, Editora. Pearson-Prantice Hall, 2001. 2a Edição.

UCHÔA, Joaquim Q. ; SIMEONE, Luiz E. ; *et al* – **Linux Intermediário** - Lavras, MG – Editora. UFLA/FAEPE – 2003 - Capítulo 6 – páginas. 115-126

UCHÔA, Joaquim Q. – **Segurança em Redes e Criptografia** - Lavras, MG – Editora. UFLA/FAEPE – 2003 - Capítulo 3 – páginas. 17-22